
# Determination of Algorithm Parallelism in NP Complete Problems for Distributed Architectures

THESIS

Ralph Andrew Beard
Captain, USAF

AFIT/GCE/ENG/90M-1





DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GCE/ENG/90M-1



# Determination of Algorithm Parallelism in NP Complete Problems for Distributed Architectures

THESIS

Ralph Andrew Beard
Captain, USAF

AFIT/GCE/ENG/90M-1

AFIT/GCE/ENG/90M-1

# Determination of Algorithm Parallelism in NP Complete Problems for Distributed Architectures

THESIS

Presented to the Faculty of the School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

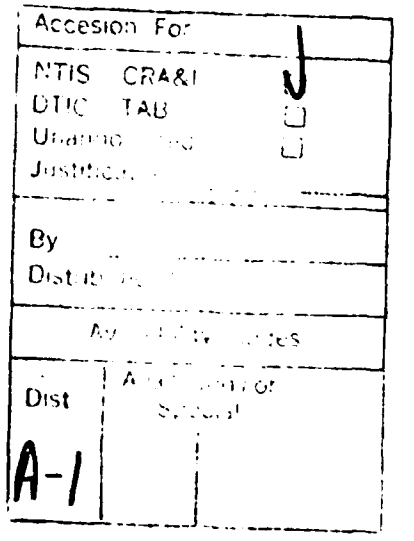
Requirements for the Degree of

Master of Science in Computer Engineering

Ralph Andrew Beard, B.S.

Captain, USAF

March, 1990



*Acknowledgments*

There are several people that I need to thank. First, my thesis advisor, Dr. Gary Lamont. He has given me more than my money's worth here at AFIT. His unrelenting drive for excellence was sometimes an irritation but mostly an inspiration. I have truly enjoyed working for and with him.

I must also thank my friend, Mike Proicou. He is the closet thing I know to a UNIX guru. He helped me on many occasions learning the intricacies of UNIX and C. I'm sure he saved me countless hours of work.

None of the above can compare to the tolerance and understanding I have received from my wife, Patty and sons, Zachary and Adam. We had heard that AFIT was tough but the rumors were too kind. Although my sons did not fully understand the cause of my absence, I don't think we lost any love for each other during my work here at AFIT. Patty was very supportive during those times when my attitude went sour and I wasn't so easy to live with. In the long run, I believe it was worth the pain.

Intel Corporation in Beaverton, Oregon was a big help during the collection of data for the graphs. AFIT's iPSC/2 computer has only eight nodes making it difficult to gather many data points for graphs. I contacted Tony Anderson, a service representative and he put me in touch with Ray Bradbury and Sean Griffin who set me up with an account on a 64 node hypercube.

Ralph Andrew Beard

*Table of Contents*

*List of Figures*

*List of Tables*

AFIT/GCE/ENG/90M-1

*Abstract*

The purpose of this research is to explore the methods used to parallelize NP-complete problems and the degree of improvement that can be realized using a distributed parallel processor to solve these combinatoric problems.

Common NP-complete problem characteristics such as a priori reductions, use of partial-state information, and inhomogeneous searches are identified and studied. The set covering problem (SCP) is implemented for this research because many applications such as information retrieval, task scheduling, and VLSI expression simplification can be structured as an SCP problem. In addition, its generic NP-complete common characteristics are well documented and a parallel implementation has not been reported.

Parallel programming design techniques involve decomposing the problem and developing the parallel algorithms. The major components of a parallel solution are developed in a four phase process. First, a meta-level design is accomplished using an appropriate design language such as UNITY. Then, the UNITY design is transformed into an algorithm and implementation specific to a distributed architecture. Finally, a complexity analysis of the algorithm is performed.

The a priori reductions are divide-and-conquer algorithms; whereas, the search for the optimal set cover is accomplished with a branch-and-bound algorithm. The search utilizes a global best cost maintained at a central location for distribution to all processors. Three methods of load balancing are implemented and studied: coarse grain with static allocation of the search space, fine grain with dynamic allocation, and dynamic load balancing.

A serial and three SCP parallel algorithms were implemented and executed on an iPSC/2 computer. Tests on large SCP problems indicate limited speedup over the serial program with the coarse grain version using static allocation and improved speedup with the fine grain version using dynamic allocation. The use of dynamic load balancing further improves the speedup and led to a super-linear speedup.

# Determination of Algorithm Parallelism in NP Complete Problems for Distributed Architectures

## *I. Introduction*

### *1.1 Problem Statement*

Many problems in AI, circuit simulation, communications, control, operations research, VLSI, and weapon-to-target assignment involve problems that reflect, in the worst case, an enumeration of all possible paths to an optimal solution; i.e., a combinatoric explosion. The associated solution-time characteristics are thus bounded by exponential functions. This family of problems is called "NP-complete" or "NP-hard" and all are transformable to each other in polynomial time. The purpose of this research is to explore the degree of improvement that can be realized in general and specifically using a distributed parallel processor to solve NP-complete problems.

The following sections described the nature of NP-complete problems in Section 1.2 followed by the research objectives presented in Section 1.3 and the scope of the research in Section 1.4. Since much of this work is software oriented, a brief explanation of applicable software engineering practices is presented in Section 1.5 with an introduction to the rest of the document contained in Section 1.6.

### *1.2 Background*

Many of the combinatoric problems mentioned in Section 1.1 are NP-complete and may be solved with search techniques developed to solve the following NP-complete problems (4, 25):

Set Covering Problem (SCP)
Assignment Problem
Hamiltonian Circuit Problem

Traveling Salesman Problem
0/1 Knapsack Problem

The SCP is generally applicable to many different problems including airline and assembly line scheduling, design of computer systems, railroad-crew scheduling, and political districting (18:591) (56:94) (7:1152). Furthermore, since the SCP is an NP-complete problem, it can be used to solve other NP-complete problems such as the assignment and graph coloring problems. The SCP is briefly described in Chapter II and a detailed explanation is available in Appendix E along with descriptions of other four NP-complete problems. These examples are typical of NP-complete problems and serial solutions to these problems are well known and documented for specific cases as well as for the general case (18, 4, 26, 17, 13, 29, 5). Furthermore, some work has been performed involving parallel solutions (56, 48, 23, 46, 50, 25). Of the previous examples, emphasis is placed on the SCP because of its general applicability to many different problems.

Combinatoric issues are becoming more visible to software developers due to the expanding complexity of problems. Problems such as optimal resource scheduling and robot arm manipulation in three dimensional space are known to be at least NP-complete and require a combinatoric search for an optimal solution (19, 24). A resource scheduling problem receiving recent attention is the problem of finding the optimal solution to the assignment of weapons to targets. A timely solution to this assignment problem is one of the many problems facing the construction of a global defense system for the Strategic Defense Initiative (SDI) (2, 3).

Consider the assignment of $w$ weapons to $t$ targets where $w < t$. It is easy to show by induction that there are $w^t$ possible assignments of weapons to targets. If costs (say based on threat, location of impact, etc.) are associated with particular weapon/target assignments, the optimal assignment can only be found by checking either implicitly or explicitly every possible assignment. Consider the best serial algorithm available to solve this problem. If only two weapons are used, $2^t$ possible assignments must be checked. If each assignment is checked in one microsecond, the problem requires $2^t$ microseconds to solve a problem of size $t$. Consider the optimal assignment of only 60 targets. If every possible assignment is checked, this problem requires $2^{60}$ microseconds or 366 centuries to

find the optimal assignment of weapons to targets (56:93). Clearly, given a large number of weapons and targets, this assignment problem is not computable within a reasonable amount of time since the time required to solve the problem in this manner grows exponentially.

Such exponential explosions in the solution time are indicative of NP-complete problems; therefore, software designers must consider alternative algorithms which solve the problem but avoid a complete enumeration of the problem space. Three such alternatives are the use of polynomial-time algorithms which yield an optimal solution, the use of probabilistic algorithms which yield an acceptable solution, or the use of more processing power applied to the optimal search.

The first alternative is to develop an algorithm which produces an optimal solution to NP-complete problems in polynomial-time (e.g., $O(n^c)$). Since the example assignment problem is NP-complete, it is sufficient to say that if there exists any NP-complete problem possessing a polynomial-time algorithm, then this assignment problem can also be solved in polynomial time (29:502). Much work has been devoted to finding such an algorithm for NP-complete problems. Since a polynomial-time algorithm has not been found, it is unlikely that NP-complete problems can be solved in polynomial time. Hence, a polynomial-time algorithm is not available to solve this assignment problem (13:337).

A second alternative is to use a probabilistic algorithm which yields an *acceptable* versus an optimal solution. Probabilistic algorithms such as Monte Carlo, numerical, Sherwood, and Las Vegas are available to solve NP-complete problems; however, such algorithms are not guaranteed to return an optimal solution (13:223). Since this thesis effort is concerned with finding the optimal solution, probabilistic algorithms are not be considered except where they might improve the optimal search procedure.

The last alternative, the substitution of more processing power, is the basis of this research; specifically, the use of many processors (multiprocessors) working simultaneously toward a solution to a common problem (parallel processing). Using multiple processors to solve an NP-complete problem does not change the combinatoric nature of the problem; however, it generally increases the size of the solvable problem (42:4).

## *1.3 Research Objective*

The objective of this research is to study the degree to which NP-complete problems can be parallelized. Key to this study are the identification of a set of common characteristics possessed by all NP-complete problems, the selection of appropriate performance metrics for comparing the various algorithms, and selection of problems or algorithms which are good representative examples of this broad class of problems. Briefly, the term *combinatoric search* implies that all the algorithms must search a potentially explosive problem space. To improve the efficiency of the search, many search techniques employ various preprocessing methods to order or reduce the input data and bounding computations to prune the search tree. The preprocessing may take the form of a simple sort of the input data or may utilize an a priori reduction algorithm which removes redundant information. By removing redundant information, the a priori reductions decrease the size of the input problem space which leads to a decrease in the size of the search graph and a corresponding decrease in the solution time. These and other common characteristics are described more fully in the following chapters.

To study the parallelization of NP-complete problems requires the selection of a representative problem which is proven NP-complete. The SCP was chosen for this research because many applications such as graph coloring, information retrieval, task scheduling, and VLSI logic expression simplification can easily be structured as an SCP problem. In addition, its generic NP-complete common characteristics are well documented, it has been studied in the general case for serial algorithms (18, 7), and a parallel implementation has not been reported. A variety of techniques have been used to solve the SCP (divide-and-conquer, branch-and-bound, dynamic programming, etc.); hence, serial algorithms are available in many forms.

## *1.4 Scope*

This thesis effort develops and implements a parallel algorithm to obtain an optimal solution for the general SCP. Since this is an optimal search, probabilistic algorithms (for example: numerical, Monte Carlo, Las Vegas, Sherwood) are not considered nor implemented. Probabilistic algorithms for solving the SCP are especially useful when the

problem is very large or the amount of time available to compute the answer is limited. However, such algorithms are not guaranteed to produce the optimum answer or even an answer. It is the intent of this research to address, to the maximum extent possible, the following goals:

1. Determine the division of control and data between the distributed processors such that maximum performance is realized. Maximum performance relates to the execution time of the parallel algorithms compared to those of a similar serial algorithm.
2. Develop appropriate performance metrics for measuring the speed and efficiency of the parallel programs.
3. Investigate the effects of static and dynamic load balancing on program performance.
4. Investigate the amount of communication versus computation occurring in the algorithms.

Since the design of parallel algorithms is such a diverse field, many items could be added to this list; however, the project must conclude within a specified time period. Hence, this list is complete with respect to the purpose of this research; namely, to study methods of parallelizing NP-complete problems and to implement a parallel algorithm for a suitable NP-complete problem.

### *1.5 Software Engineering Practice*

Much of this effort focuses on the design and development of a parallel solution to the SCP which is implemented in software on a parallel computer; hence, a structured approach is required in keeping with good software engineering practices. Therefore, the general data and control structures are developed first using top-down design and a structured design syntax. The general data and control structures are then mapped to a parallel architecture.

Various structured and unstructured methods and tools are available to assist software engineers in their design task:

1. Pseudo-English
2. Structured Analysis and Design Tool (SADT) (49:192)
3. Unbounded Nondeterministic Iterative Transformations (UNITY) (16:6)
4. Structure Charts
5. Abstract Data Types (ADTs) (30:7)

For this research, a structured design method enabling the expression of parallel and serial operations is required. From the above list, UNITY is specifically designed for parallel programming tasks and structure charts are an easily understood method of representing communication.

UNITY is a design syntax developed by Chandy and Misra for use in developing parallel programs. It is their attempt to incorporate a formal syntax into the parallel program design process and is similar, in many respects, to the Hoare's (28) method of designing concurrent sequential processes. Both methods are based on predicate calculus and structured design methods. Further details of UNITY as well as other parallel algorithm design methods are left to Section 2.5.3. Other authors such as Jamieson (38), Carriero (15), Wah (56), and Fox (25) describe similar though less formal design methods. Each author's particular design methods undoubtedly influenced the design of the algorithms developed for this research; however, UNITY programs are presented since they are more formal.

The valuable insight gained from the UNITY program design process assists in the development of pseudo-code algorithms for the parallel SCP programs. The pseudo-code algorithms are developed in Chapters III and IV and include references to the structure charts and ADTs contained in Appendices A and B.

The control and data structures of the SCP are structured so that the algorithms are easily changed to investigate different methods of load balancing. A serial version of the SCP is developed first followed by three parallel versions. The first parallel implementation is the simplest, utilizing basic methods to distribute the data among all the processors. The second implementation is only slightly more complicated and is actually a modification of the first. The second implementation distributes the initial data dynamically between the processors. The third and final parallel implementation is the most complex and efficient.

It adds a dynamic load balancing capability to the search process. As the processors complete their individual search, they poll other processors in an attempt to share the remaining work. Thus, all three parallel implementations are built on previous solutions to the problem.

Since many parts of the parallel SCP solution involve serial algorithms, these algorithms are developed and executed on a personal computer using Borland's Turbo C programming environment (11, 12). Borland's programming environment integrates an editor, compiler, and symbolic debugger into one cohesive package, thus permitting quick development and debugging of software. Although Turbo C operates on a serial computer, the parallel programs are compiled and linked with Turbo C to eliminate the obvious compile-time errors. These parallel programs are not executed on the personal computer because of the large amount of code needed to simulate the parallel communications.

### *1.6 Summary*

To find the optimum solution to an NP-complete problem requires exponential time in the worst case. Hence, large problems quickly become incomputable within a reasonable amount of time. The objective of this research is to investigate the amount of parallelism inherent in these "hard" problems. The SCP is a good representative of the class of NP-complete search problems since many NP-complete problems can be structured as an SCP problem and it incorporates many of the methods typically used in combinatoric searches; namely, a branch-and-bound search, dynamic programming, easily computed lower bound, and polynomial-time computable a priori reductions.

To limit the physical size of this document, assumptions are made concerning the reader's breadth of knowledge. Subjects such as NP-completeness, parallel architectures, and parallel algorithms are presented but not in detail. Bibliographic entries are given to more complete references. No assumption is made concerning the reader's knowledge of the SCP; hence, a detailed explanation is available in Appendix E along with examples illustrating its application to other NP-complete problems and 'real-world' applications.

The next chapter, Chapter II, is the background and requirements review. Many

hours were spent researching the various aspects of NP-complete problems, search techniques and algorithms, parallel architectures, and the SCP; hence, Chapter II is somewhat lengthy. Readers familiar with basic search techniques should probably skip Section 2.4. Sections 2.5 and 2.6 should be read by most readers as they describe parallel architectures, parallel algorithms, and parallel algorithm design for NP-complete problems. Section 2.3 is a brief description of the SCP and is supplemented by Section E.6 in Appendix E.

The design of the parallel algorithms is conducted in two stages. Chapter III describes the methodology and preliminary design used to solve this complex problem; whereas, Chapter IV is a detailed design of the programs for the SCP. Much of Chapter IV is mainly dedicated to the construction of UNITY representations of the SCP and verbal descriptions of the algorithms. On the other hand, Chapter IV presents many detailed pseudo-code algorithms for the SCP routines. Chapter V consists of a test plan to test the SCP algorithms and explains how to read the results obtained from parallelizing the SCP. Chapter VI contains an interpretation of a the results, a section on lessons learned, and a list of recommendations for further research.

# *II. Background & Requirements*

## *2.1 Introduction*

An understanding of NP-completeness, search techniques and algorithms, parallel architectures, parallel software design, and the SCP are essential to the completion of this research. As such, this chapter is divided into five main sections: NP-completeness, the SCP, generic search techniques, parallel architectures, and parallel software design. Each subject area has been extensively studied by other authors and many books and articles are available. It is not the intent of this chapter to conduct an exhaustive review of each subject; rather, this chapter summarizes and expands on those areas which are most beneficial to the focus of this research. Since the terminology is not well defined, this thesis takes the view of Fox and others (25:477) and treats *concurrent* and *parallel* as synonymous terms.

The next section, Section 2.2, is an introduction to the theory of NP-completeness followed by a brief description of the NP-complete problem to be parallelized in this research, the SCP. Section 2.4 is a review of common, generic search techniques used to solve many NP-complete problems. Parallel architectures and algorithms are discussed in Section 2.5 and Section 2.6 integrates the information in the previous sections to discuss parallel algorithm design for NP-complete problems.

## *2.2 NP-Complete Problems*

"It is an unexplained phenomenon that for many of the problems we know and study, the best algorithms for their solution have computing times which cluster into two groups" (29:501–502). The solution time for the first group of problems is bounded by a polynomial-time function. For example, sorting — $O(n \log n)$, binary searching — $O(\log n)$, and matrix multiplication — $O(n^{2.81})$. The second group of problems are those whose best known algorithms are nonpolynomial. For example, the TSP — $O(n^2 2^n)$, 0/1 knapsack problem — $O(2^{\sqrt{2}})$, and the SCP — $O(2^n)$ (29:501–502). The thrust of this master's thesis is of course the second class. More specifically, a collection of problems in the second class termed nondeterministic polynomially-complete (NP-complete).

All NP-complete problems have two distinguishing characteristics. First, an NP-complete problem must be in the class $\mathcal{NP}$. Secondly, any NP-complete problem must be transformable to all other NP-complete problems in polynomial time and vice-versa (4:373). Problems in $\mathcal{NP}$ consist of all problems which can be solved in polynomial time on a nondeterministic turing machine (NDTM). An NDTM can solve an unbounded number of problems in parallel; therefore, it can solve both polynomial-time and nonpolynomial-time algorithms in polynomial time (26:12). Clearly, such a machine does not exist; hence, problems in $\mathcal{NP}$ whose best know algorithms are nonpolynomial can not be solved in polynomial time.

The second characteristic of NP-complete problems is that they must be transformable to each other in polynomial time. Hence, given an NP-complete problem that can be solved in polynomial time, *any* NP-complete problem can be solved in polynomial time (29:502). This last statement has profound implications! There are many problems whose best known algorithms are bounded by exponential functions. Moreover, many of these same problems have been proven to be NP-complete; that is, they are in $\mathcal{NP}$ and are transformable to each other in polynomial-time. The problem is that no one has yet proven that any of these NP-complete problems can not be solved in polynomial time. However, given the current algorithms, it is simple to show that large instances of these problems are not solvable by any computer since the time required to compute the solution grows exponentially (13:337).

### *2.3 The Set Covering Problem*

The set covering problem (SCP) is one of a large class of NP-complete problems[1] extensively studied in the late 1960's and early 1970's in connection with operational research problems. The SCP is the problem of finding the minimum number of columns in a 0–1 matrix[2] such that all rows of the matrix are covered by at least one element from any column and the cost associated with the covering columns is optimal (minimum or

[1]See Aho (4:392) for a proof of the SCP's NP-completeness

[2]A 0–1 matrix is a rectangular matrix in which a covered row is denoted by a '1' in the covering columns. If the rows in the matrix represent the vertices of a graph, the existence of an arc between any two vertices is denoted by a '1' in the column of the matrix.

maximum) (17:39). As an example, Figure 2.1 shows a 0–1 matrix in which the rows are covered by several different combinations of columns. Columns 0, 1, 2, 3, and 4 form a cover with a total cost of 27. The optimal cover is columns 0, 3, and 4 with a cost of 15.

| | | Sets | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| Vertices | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 |
| | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 |
| | 2 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| | 3 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 1 |
| | 4 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 |
| | 5 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 |
| | | 4 | 7 | 5 | 8 | 3 | 2 | 6 | 5 |
| | | Costs | | | | | | | |

Figure 2.1. 0–1 Matrix for a Set Covering Problem (17:54)

As stated previously, the SCP has applications in solving many 'real-world' and NP-complete problems. Examples of which are presented in Section E.6.2. Since much of this research revolves around the SCP the reader is encouraged to at least scan the detailed explanation of the SCP presented in Appendix E. The process of solving the SCP, or any other NP-complete problem, typically requires an optimal search of a problem space. That is, a search of the problem space so that some value is minimized (or maximized) subject to some constraints. In the SCP, the cost of the cover is minimized subject to the constraint that all rows must be covered by elements in at least one column.

## *2.4 Search Techniques*

Numerous search techniques have been developed and each technique has countless variations which depend on the specific problem to be solved. The following sections explain six common generic search techniques: the greedy method, a brute-force search, divide-and-conquer, dynamic programming, backtracking, and branch-and-bound.

*2.4.1 Greedy Method* Consider the search for a cover in the 0–1 matrix of Figure 2.1 based strictly on a greedy method. The greed method chooses, at each step in the

search, the *best* column for inclusion in the cover without considering the future. Columns included in the cover are never removed and those excluded from the cover are never reconsidered (13:80). The selection criteria for the *best* column varies depending on the problem. For instance, it may be the lowest (or highest) cost column or the column with the most elements (1's). Such a search is not guaranteed to be optimal but it is simple and usually quite fast. For the example matrix, a search based on the greedy method might choose columns {0, 1, 2, 3, 4} since they comprise an extremely simple cover. In this case, *best* is defined as 'the next column'.

*2.4.2 Brute-force Method* One of the most straight forward approaches to solving any search problem is the brute-force approach. "Suppose $m_i$ is the size of a set $S_i$. Then there are $m = m_1, m_2, \ldots, m_n$ n-tuples which are possible candidates for satisfying the function $P$. The brute force approach would be to form all the n-tuples and evaluate each one with $P$, saving those which yield the optimum" (29:323–324). For example, let $m_i$ = 3. The complete list of sets for $S_i$ is {{1}, {2}, {3}, {1, 2}, {1, 3}, {2, 3}, {1, 2, 3}}. The brute force method checks every possible combination of sets for a solution. Hence, $m_i$ = (3, 4, 5, or 6) would require that 7, 15, 31, and 63 combinations of sets be examined. Clearly, the number of sets to check increases exponentially with $m_i$.

The most common brute force methods are the depth-first search and the breadth-first search. Both are best explained with the aid of a tree diagram; hence, in the following sections, an explanation of the search is first given followed by an example. The examples are based on the search for a cover in the matrix of Figure 2.1. The nodes of the tree represent the columns of the matrix and, when combined, form a cover of the rows.

*2.4.2.1 Depth-first Search* This generic search vertically expands a search tree. Consider an undirected graph composed of $N = \{n_1, n_2, \ldots, n_p\}$ nodes and connecting links where the semantic representation of the nodes and links is determined by the problem to be solved. For instance, the nodes might represent the processes in an operating system and the links might represent dependencies between the processes. Any two nodes of the graph are termed adjacent if they share a common link. "To carry out a depth-first traversal of the graph, choose any node $\gamma \in N$ as the starting point. Mark

this node to show that it has been visited. Next, if there is a node adjacent to $\gamma$ that has not yet been visited, choose this node as a new starting point" and repeat the search process (13:171). For an optimal search, any combination of nodes that represents a feasible solution to the problem is compared against a previously obtained best feasible solution with the better solution retained for future comparisons. The search continues until all nodes have been visited and all solutions have been found. By completely exploring one subtree before progressing to the next, this search technique quickly finds an initial solution; however, it must still expand every node in the search tree to ensure that the optimal solution is found (23:1500).

Consider a depth-first search of the 0–1 matrix given in Figure 2.1. A small portion of the search tree is shown in Figure 2.2. The search first chooses column 0 to add to the solution. Since column 0 is not a cover, it is not retained. It then combines the following columns {0,1}, {0,1,2}, {0,1,2,3}, {0,1,2,3,4}. The last combination is saved since it covers all the rows. The search continues to enumerate the rest of the combinations: {0,1,2,3,4,5}, {0,1,2,3,4,5,6}, {0,1,2,3,4,5,6,7}. Obviously, these combinations are no better than the first cover found; therefore, they are not retained. Since the depth-first search can go no deeper down this branch, it retraces its steps until it can go forward again. The search continues with {0,1,2,3,4,5,7}, {0,1,2,3,4,6}, and so on until all possible combinations of columns have been formed. As stated, an initial cover is quickly found and then used to compare against future covers and each branch is completely searched before proceeding onto the next branch. It is quite obvious that this search is effective but could be much more efficient. Methods to improve the efficiency are discussed in future sections.

*2.4.2.2 Breadth-first Search* In contrast to the depth-first search, this generic search expands the search tree horizontally. "When a breadth-first search arrives at some node $\gamma$, it first visits all the neighbors of $\gamma$" before visiting any of the nodes adjacent to $\gamma$ (13:182). In other words, each level of the tree is expanded before proceeding to the next level. The search is just as exhaustive as the depth-first search and the optimal solution is only found when all nodes have been expanded.

Figure 2.3 shows a portion of the breadth-first expansion of the 0–1 matrix. It first

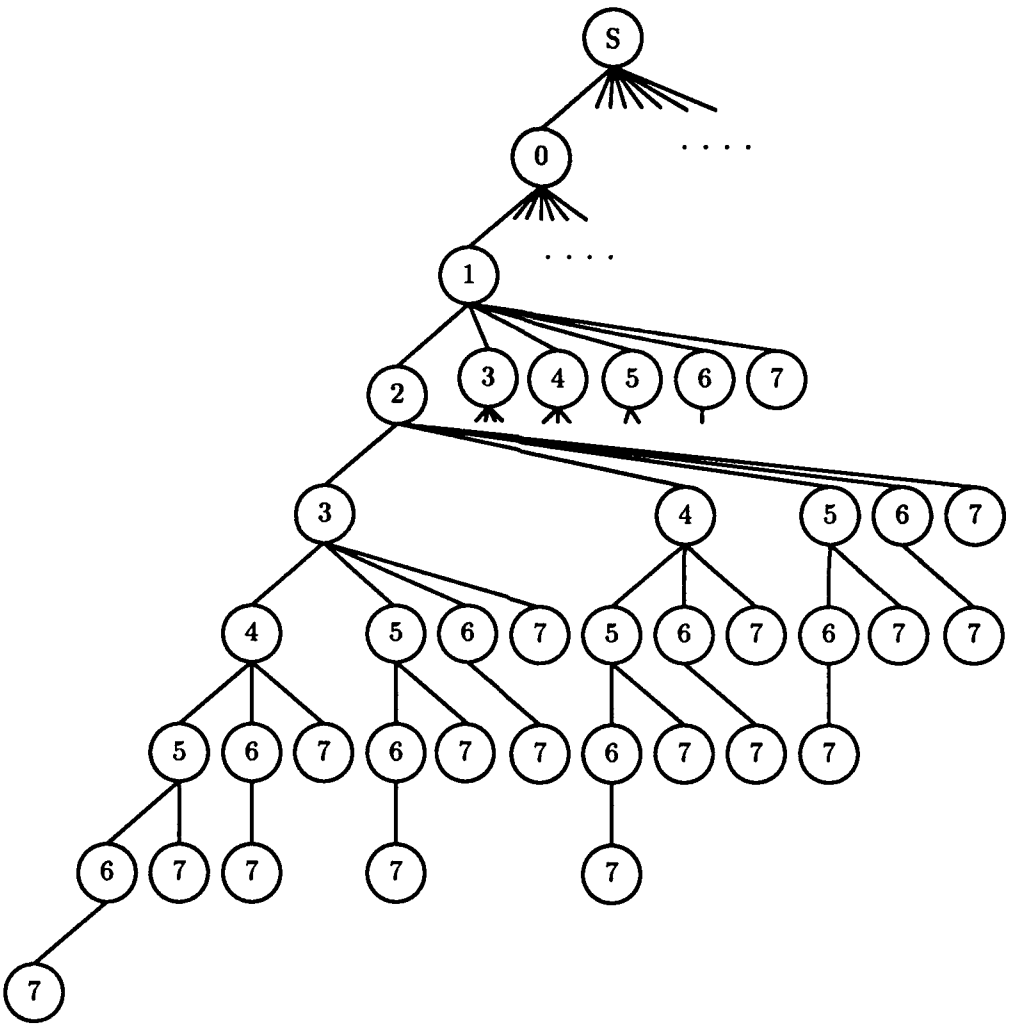



Figure 2.2. Depth-first Search of Figure 2.1

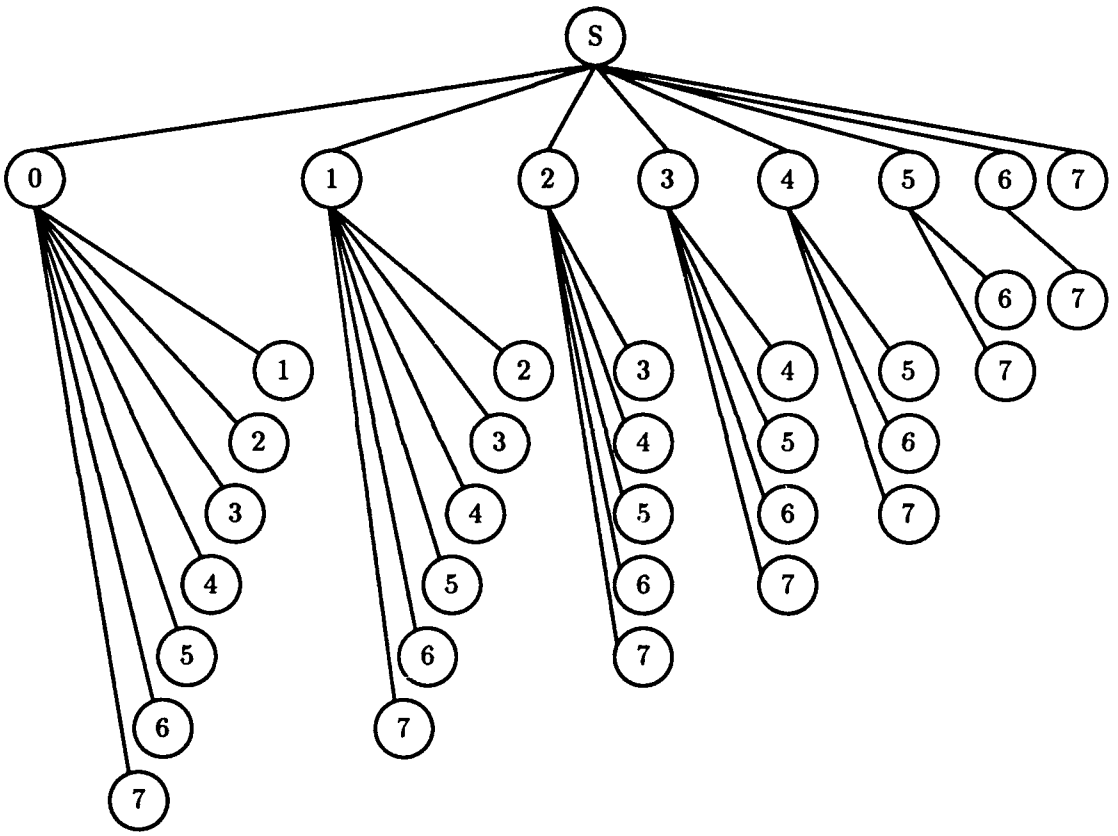



Figure 2.3. Breadth-first Search of Figure 2.1

visits nodes 0, 1, 2, 3, 4, 5, 6, and 7 in that order and then returns to node 0. Node 0 is expanded one additional level as are nodes 1, 2, 3, 4, 5, and 6 in order. Node 7 is not expanded any further since it is the last node in the search. As with the depth-first search, it easy to see that this generic search method is effective but far from efficient.

*2.4.3 Divide-and-Conquer* Divide-and-conquer is a top-down approach to searching. The basic premise is to continually divide the problem into smaller subproblems until, at some point, the subproblems can be efficiently solved by a simpler method. Once the subproblems are solved, they are combined until the original problem has been solved (13:142). The efficiency of a divide-and-conquer approach depends on the method used to solve the subproblems and the method by which the subproblems are recombined (13:105). A binary search is a simple application of the divide-and-conquer approach to a problem solution (13:109). Given a sorted list, the list is continually divided in half until the algorithm converges upon the item or determines the item is not present.

*2.4.4 Dynamic Programming* "An algorithm design method that can be used when the solution to a problem may be viewed as the result of a sequence of decisions" (29:198). The underlying goal of dynamic programming is to avoid calculating the same thing twice by making explicit appeal to the Principle of Optimality (13:142). "This principle states that an optimal sequence of decisions has the property that whatever the initial state and decision are, the remaining decisions must constitute an optimal decision sequence with regard to the state resulting from the first decision" (29:199). However, not all problems conform to the Principle of Optimality. But if the principle applies, the number of decisions are reduced since the solution to previous optimal decisions is retained and used in future decisions and suboptimal decisions are not considered.

The Principle of Optimality is usually formulated as a recurrence relation using either a forward or backward approach:

> Let $x_1, x_2, \ldots, x_n$ be the variables for which a sequence of decisions has to be made. In the *forward approach*, the formulation for decision $x_i$ is made in terms of optimal decision sequences for $x_{(i+1)}, \ldots, x_n$. In the *backward approach* the formulation for decision $x_i$ is in terms of optimal decision sequences for

> $x_1, \ldots, x_{(i-1)}$. Thus, in the forward approach formulation we "look" ahead on the decision sequence $x_1, x_2, \ldots, x_n$. In the backward formulation we "look" backwards on the decision sequence $x_1, x_2, \ldots, x_n$. (29:201)

Many examples of dynamic programming are given in the literature (13, 29, 45, 9) and most are quite lengthy; hence, no examples are presented here. However, a dynamic programming algorithm is employed to calculate a lower bound for the SCP and is presented in Section E.6.7.

*2.4.5 Backtracking* Backtracking algorithms are a variation on the basic tree search algorithm and carry out a systematic search of the problem space (13:185). "In order to apply the backtracking method, the desired solution must be expressible as an n-tuple $(X_1, X_2, \ldots, X_n)$ where the $X_i$ are chosen from some finite set $S_i$" (29:323). As each $X_i$ is added to the n-tuple, the n-tuple is checked to see if it represents a solution to the problem. At some point during the expansion, it may be desirable to backup to a previous state in the search. The n-tuple, which represents the current state of the search, facilitates an easy recall of the previous states.

Once again, consider a search of the matrix of Figure 2.1. An example backtracking search is shown in Figure 2.4. The search starts as a depth-first search until it reaches the first cover {0,1,2,3,4}. At this point, the search backtracks since a better solution can not be found down this path. The algorithm backtracks to a previous state and proceeds forward expanding adjacent nodes. Notice, however, that the algorithm does not routinely backtrack to the top of the search tree; rather, it searches subtrees before backtracking to other subtrees. New covers are compared against the previous cover and the better cover is kept for future comparisons. It is easy to see how this search is closely related to the depth-first search but is more efficient in the vast majority of problems since it does not enumerate all possible solutions as was the case with the depth-first traversal.

*2.4.6 Branch-and-Bound* The solution to NP-complete problems presented in this thesis all require a search of the state space and, as previously stated, the state space of an NP-complete problem increases exponentially as the problem size increases. Clearly, an unguided search of such a state space may continue indefinitely. The branch-and-bound

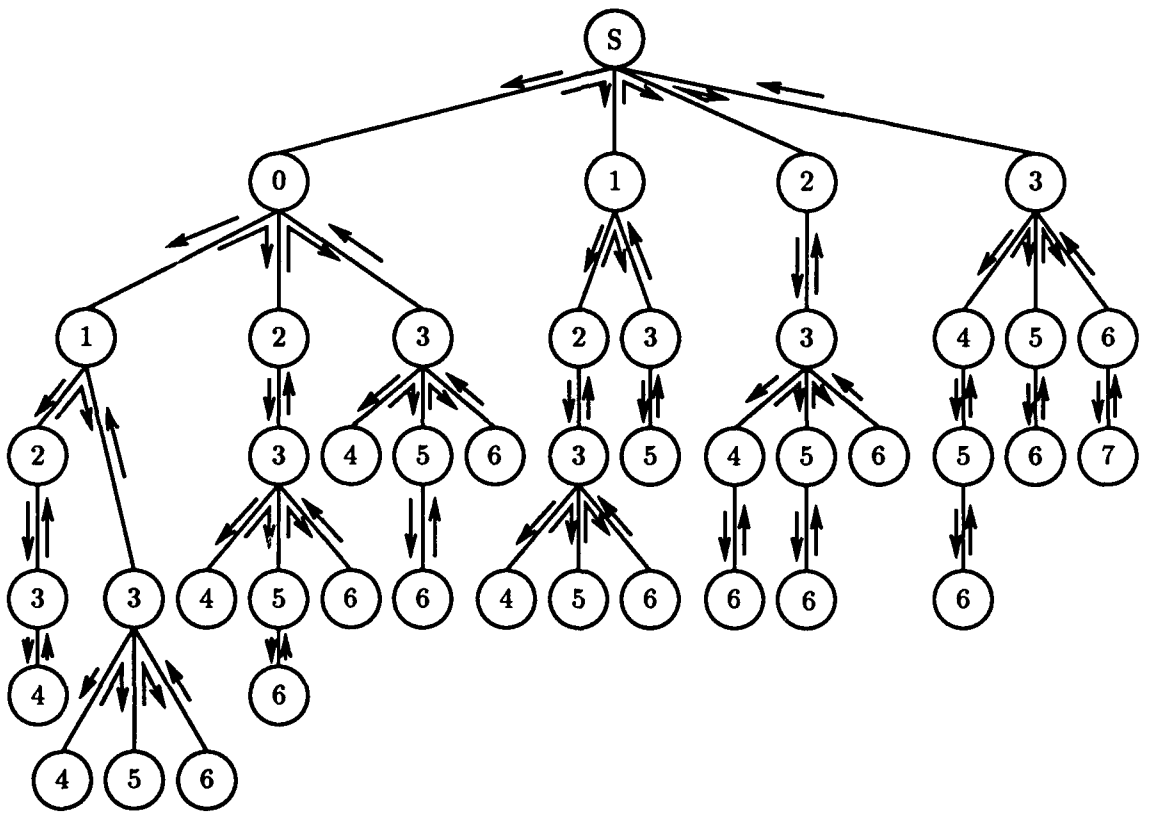



Figure 2.4. Search Using Backtracking

search is the most general of several guided, optimal searches (1:1492). It is a variation on backtracking in which the amount of searching is pruned or bounded through the use of a bounding function. At each state of the search, the cost of the current state is compared to the cost of the best solution obtained thus far. If the current state's cost is worse than the best solution obtained, the algorithm backtracks since any additional states in this branch of the search tree can only lead to a solution worse than the solution already obtained (13:199).

A "branch-and-bound algorithm consists of four major components: a *selection* procedure, a *branching* procedure, an *elimination* procedure, and a *termination* test procedure" (1:1494). The *selection* procedure selects the next state of the search to be included based on some heuristic (criteria) function such as a depth-first or breadth-first traversal. The *branching* procedure examines the next state chosen by the selection procedure and further divides the state space into smaller subproblems. The *elimination* procedure examines the subproblems developed by the branching procedure and eliminates those subproblems that can not possibly lead to an optimal procedure. And finally, the *termination* test procedure examines the subproblems produced by the branching procedure and eliminates those subproblems that do not lead to a feasible solution (1:1494).

It becomes apparent, with some thought, that many guided searches described in the literature (A*, AO*, alpha-beta) may be described in terms of a branch-and-bound search (47). Each of these guided search algorithms includes the four major components with some minor modification added to improve the search for a specific type of problem (1:1493).

*2.4.7 Improvements to Optimal Search Techniques* Except for the greedy method, all search techniques presented provide a basic structure for solving an optimal search problem. In many instances, the efficiency of the search techniques may be improved through the use of precomputation or preconditioning in the form of a priori reductions and selective bounding functions (13:205).

*2.4.7.1 Reduction Methods* In many problems, it is possible to reduce the amount of searching required with problem specific reduction techniques or precomputation to reduce the dimensions of the original graph or tree (13:211). One such reduction is to remove any states which are included in every branch of the search tree. For example, in the problem of Figure 2.4, the element corresponding to node 3 is in every solution path. Therefore, it is not necessary to include this node in every search path. Rather, the node is removed from from the input problem and retained for later insertion into the final solution. Details of more reductions are presented in Section E.6.4.

*2.4.7.2 Dominance Testing* Dominance testing is a precomputation method which may decrease the size of the search tree by comparing the current state of the search to previously saved states. For instance, if the current state is a subset of a previous state and the current state's cost is greater than or equal to the previous state's, then the algorithm can backtrack. This technique requires a list of previous states be maintained in some suitably arranged manner (list) to allow an efficient comparison to the current state. If desired, all previous states may be saved; in which case, this approach resembles a breadth-first search of the problem space. As with most engineering problems, some tradeoff must be made between the number of stored previous states and the computation time required to check for dominance (18:594–595). A good example of dominance testing, as it relates to the SCP, is presented in Section E.6.6.

*2.4.7.3 Lower Bound Computation* In addition to dominance testing, the computation of a lower bound is sometimes useful in bounding the search. A lower bound is the lowest possible cost down a branch of the search tree. Whether or not the lower bound can actually be obtained is irrelevant. What does matter is that if the current cost plus the lower bound exceeds the best cost obtained thus far, the algorithm backtracks. As the computation of the lower bound becomes more accurate, more branches of the search tree are pruned. In the best case, the lower bound is exact and the search proceeds down the optimum path without backtracking. It is natural to assume that the precision of the lower bound computation is inversely proportional to the amount of computation time required to compute the lower bound. That is, a precise lower bound may require a long

time to compute. Therefore, a suitable lower bound computation is one in which the time required to compute the bound does not adversely impact the overall search time (13:200–202) (18:595). A dynamic programming algorithm for computing the SCP lower bound is presented in Section E.6.7.

In summary, many of the preceding search techniques are used in some form or another to solve NP-complete problems; hence, derivatives of these techniques are used in this research. The objective of this research is to investigate and develop parallel search algorithms for solving NP-complete problems on parallel architectures. Before any of the above search techniques are parallelized, it is necessary to acquaint the reader with the characteristics of parallel architectures and parallel algorithm design. The next section discusses parallel architectures; specifically, the characteristics of parallel hardware and software.

## *2.5 Parallel Architectures*

"The driving force behind parallel architecture design has been the need to execute time critical or large scale applications as fast as possible" (55:165). Common methods used to achieve improved computer performance are better algorithms, code optimization, improved serial architectures, and faster technology (42:1352). With the technology of modern computers nearing the "limits set by the speed of light and quantum physics effects, there is general agreement that the only route to significantly increased performance is through *concurrent computation* — the use of many computers together to solve the same problem" (25:2). An understanding of parallel architectures and perhaps more importantly, the design of parallel programs, it required to design and implement the parallel programs for this research. The following sections are designed to do just this; namely, acquaint the reader with parallel computers and parallel algorithm design. First, three methods used to classify parallel computers is presented in the next section, Section 2.5.1, followed by a description of a cube-connected architecture in Section 2.5.2 and the characteristics of parallel program design in Section 2.5.3.

*2.5.1 General Architecture* There exist many methods to classify computers in general and parallel computers in particular. In 1966, Flynn (24) presented a simple classification scheme for computers based on their instruction and data streams as either SISD, SIMD, MISD, or MIMD (Single or Multiple Instruction stream, Single or Multiple Data stream). An SISD computer is essentially a Von Neumann, sequential computer. An MIMD computer is typically composed of several SISD computers (independent processors) connected by an interconnection network with each processor capable of executing its own independent program.

Another method used to classify parallel computers is to consider the interconnection network. In many parallel programs, the independent processors must communicate control/data information to neighboring processors. This is accomplished via the interconnection network. Popular interconnection networks are the common bus, the shuffle-exchange, the two-dimensional mesh, and the cube as shown in Figure 2.5 (25:23). Each

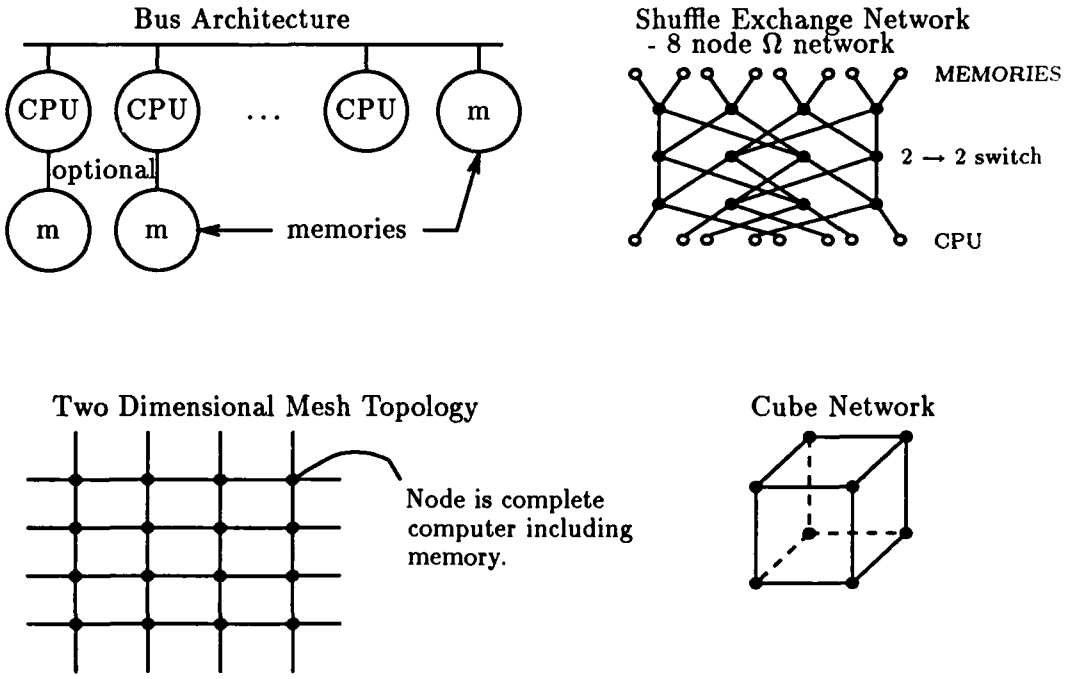



Figure 2.5. Popular Interconnection Networks (25:23)

network has characteristics which make it appealing to a particular class of problems; hence, the selection of an efficient parallel architecture is largely driven by the application.

The cube network is of primary importance to this thesis and is discussed in the following section. A good description of the other networks is available in Stone (55) and Hwang and Briggs (31).

Structure of the memory is also used to classify parallel computers. For instance, shared memory architectures consist of a large global memory. Each processor uses this memory to access common data and to pass information to other processors. Such an architecture is referred to as tightly coupled. Local memory or a loosely coupled architecture is composed of processors with their own private memory. In such a system, data and information is passed between processors via the interconnection network (14:13). In contrast to an architecture based on a global memory, the cube network is a loosely coupled architecture.

*2.5.2 Cube-connected Architecture* A cube-connected architecture derives its name from the interconnection pattern associated with the individual processors as illustrated in Figure 2.6 which depicts a cube of dimension 3. This architecture consists of $n = 2^k$

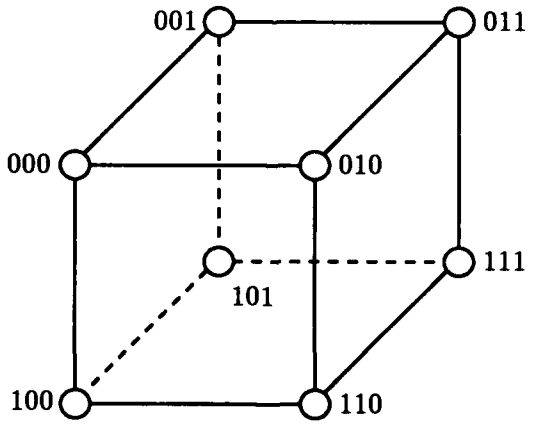



Figure 2.6. Three Dimensional Cube Network

processors labeled $0, 1, \ldots, 2^k - 1$ where $k = 3$ in the figure. If the node labels are represented in binary format, neighboring or adjacent nodes differ by exactly one bit position in their labels (50:28). As one might expect, the most efficient communications occur between adjacent nodes since the communications are not required to traverse intermediate

nodes. For example, node 000 can only communicate with node 111 after connecting with the intermediate nodes 010 and 011 or some other similar combination of nodes.

As stated previously, each parallel computer network or architecture has characteristics which make it suitable for particular applications. The cube network is desirable for tree search type applications since the nodes of the network are easily configured as the nodes of a binary tree. Hence, a cube should work well on search algorithms such as those needed to search NP-complete problems.

*2.5.3 Characteristics of Parallel Algorithms* Some authors argue that the development of parallel algorithms is the most challenging aspect of parallel computing since many factors jointly influence a parallel design and the science of parallel computing is still relatively young (25:25). The literature seems to reflect that the following are the most important issues associated with the design of parallel algorithms:

1. *Discovering the problem's inherent parallelism* (14:19). Many problems exhibit parallelism in their control structure with the presence of nested **for** loops or separate processes. As an example, the inner most **for** loops of a program may be partitioned to separate processors to work on separate parts of the data. The outer most **for** loop then controls the number of times each processor executes its **for** loop. Consider the simple task of computing the row sum of all the rows of a matrix. A parallel version of the algorithm assigns each row to one of $N$ processors which compute the sum for the individual rows as shown below. Following termination, each processor contains the sum of one row.

```
for i = 1 to N
      A[i,0] = 0.0
end for

for i = 1 to N
      parbegin
            for j = 1 to N
                  A[j,0] = A[j,i] + A[j,i]
            end for
      end parbegin
end for
```

2. *Decomposing the data to the appropriate grain size.* Grain size refers to the size of the problem allocated to a processor. The proper grain size is one which allows the parallel computer to reach its optimal performance (56:95). As expected, there is a trade-off: designing an application using many small programs (small grain) increases the amount of parallelism possible at the cost of increased communications overhead; whereas, large grain programs have less communication and more computation (42:422). Clearly, the best grain size depends not only on the application (i.e., sorting, searching) but also on the problem (i.e., sorting partially ordered sets). Hence, it is nearly impossible to find the optimal grain size and many parallel designs simply settle for the grain size that suits the algorithm design or that produces satisfactory results.

3. *Mapping and scheduling of tasks or problems.* The mapping problem is the problem of making a static assignment of processes or problems to processors (38:304). Whereas, the scheduling or processor allocation problem "is one of assigning the tasks of a parallel program among the processors or a parallel program in a manner which minimizes interprocessor communication costs while simultaneously maintaining computational load-balance among the processors" and is known to be NP-complete; hence, suboptimal solutions are generally sought (22:21).

4. *Balancing the load (static or dynamic)* (55:166). Static load balancing distributes the data among the processors before computation starts; whereas, dynamic load balancing distributes the data as the processes are executing (25:8).

5. *"Relation of the problem topology (architecture) to that of the problem"* (25:2–3). This characteristic was presented briefly in Section 2.5.1. Certain problems exhibit information structures that can be implemented in the parallel computer's interconnection network. For instance, the cube network works efficiently for many search algorithms since these algorithms use a tree structure to implement the search. On the other hand, the mesh network resembles a two-dimensional grid and fits easily into many flow problems such as modeling heat flow.

No doubt there are more issues associated with the design of parallel algorithms. This list is simply a representative sample.

The ultimate goal in any parallel algorithm is to obtain a speedup proportional to the number of processors over the best serial algorithm (56:93). In fact, such a measure is a key performance indicator given by the following equation:

$$S(N) = \frac{t_{serial}}{t(N)}$$

where $S(N)$ represents the speedup, $t_{serial}$ is the execution time of the serial program, and $t(N)$ is the execution time of the parallel algorithm on $N$ processors. As simple as this measurement seems, it is not always a reliable indicator of an algorithm's performance. In many serial algorithms, computational analysis has derived an acceptable lower bound (e.g., $n\log(n)$ for a sort (29:350)). The same statement does not hold for most parallel algorithms. The analysis of parallel algorithms is much more complex "since many factors jointly determine the system performance and the modification of some factors affects many others" (42:421). A number of possible parameters to evaluate include (53):

- Absolute execution time.
- Local and global memory requirements.
- Effect of varying granularity on message traffic.
- Synchronization between processes or processors.
- Contention for global data.
- Idle time — time spent by a processor working on the problem versus the time spent trying to retrieve a problem.

Methods of analysis for parallel algorithms vary, but a common pattern seems to hold. The analyst first specifies the number of processors in the parallel computer (e.g., $n$ or $n^2$ processors), then constrains the problem to be analyzed, and finally specifies the interconnection network. An order-of analysis is then obtained based on the specified real or fictitious architecture.

Aside from the difficulties of defining acceptable performance measures and analyzing the parallel algorithms, Jamieson, Gannon, and Douglass have suggested the existence of parallel programming paradigms and have identified three parallel computational models: "1) Compute-aggregate-broadcast algorithms, composed of a compute phase, a combining phase, and a broadcast phase in which the aggregate information is returned to the processes; 2) Pipelined and systolic processes; 3) Divide-and-conquer strategies" (38:1). In addition to these computational models, in Chapter 3 of (38), Jamieson presents the algorithm development life cycle depicted in Figure 2.7.

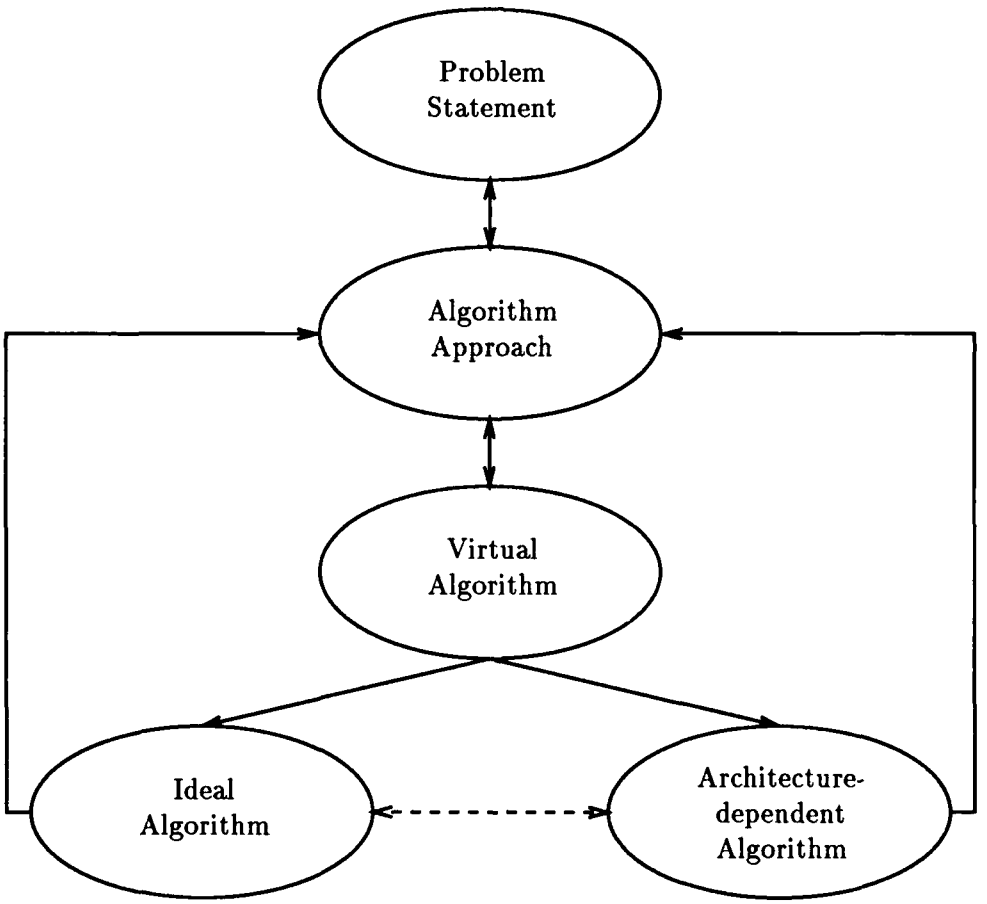



Figure 2.7. Algorithm Development Life Cycle (38:69)

The first step in the development cycle is to develop a problem statement and general approach. Next, a "virtual algorithm" is developed which "represents the point at which

the computational steps to be performed have been determined, but the mapping of those steps to an architecture has not yet been committed." From the virtual algorithm, the designer develops an ideal algorithm which corresponds "to the parallel algorithm that would be selected if no constraints were placed on the architecture configuration. In general, this is the best known parallel implementation of the virtual algorithm." Given the virtual and ideal algorithms, a choice of a physical architecture is made and the "architecture-dependent algorithm" is developed. Jamieson's development life cycle does not include feedback paths, but it is the belief of this author that no engineering development can proceed without at least revisiting previous decisions at each step in the development. Therefore, Figure 2.7 includes feedback paths not present in Jamieson's original model.

Jamieson's development of a parallel software life cycle follows the design process noted in at least two other articles. In their article (56), Wah, Li, and Yu first describe the characteristics and problems associated with the parallelization of combinatorial search problems (problem statement). They then discuss the types of algorithms used to solve these search problems (general approach to the problem). Rather than develop a virtual and ideal algorithm for the search problems, they propose a computer specifically designed for combinatorial search problems called "MANIP — a multiprocessor for parallel best-first search" (56:97).

Carriero and Gelernter (15) present yet another development cycle for parallel programming which is similar to Jamieson's model. Their model develops *conceptual classes* for understanding parallelism and *programming paradigms* for implementing parallel programs.

> The conceptual classes are *result parallelism*, which centers on parallel computation of all elements in a data structure; *agenda parallelism*, which specifies an agenda of tasks for parallel execution; and *specialist parallelism*, in which specialist agents solve problems cooperatively. The programming paradigms center on *live data structures* that transform themselves into result data structures; *distributed data structures* that are accessible to many processes simultaneously; and *message passing*, in which all data objects are encapsulated within explicitly communicating processes. (15:323)

Their development process also follows the general structure presented. "To write a parallel program, (1) choose the concept class that is most natural for the problem; (2) write a program using the method that is most natural for that conceptual class; and (3) if the resulting program is not acceptably efficient, transform it methodically into a more efficient version by switching from a more-natural method to a more-efficient one" (15:325).

Chandy and Misra (16) pursue a different approach to the development of parallel algorithms. They assert that "the unity of the programming task transcends differences between the architectures on which programs can be executed and the application domains from which problems are drawn" (16:vii). Based on this assertion, they develop a computational model with an associated "notation for specifications" called UNITY (Unbounded Nondeterministic Iterative Transformations) which is based on a state-transition model of a problem. Their design process is to state the problem and then iteratively develop more complex UNITY representations of the problem through program schemas until the UNITY metaprogram is sufficiently developed to map directly to a target architecture. This process of iteratively developing more complex representations of the problem is basically the same approach proposed in the previous paragraphs. The key contribution made by Chandy and Misra is the development of a design syntax which can be used to define the algorithm at a high level. Since UNITY is more structured, the parallel algorithms designed in Chapters III and IV are stated in UNITY along with an explanation of the semantics associated with the syntax. It is worth mentioning that UNITY is essentially a variation of predicate calculus and is similar to the notation developed by Hoare in *Communicating Sequential Processes* (28).

### *2.6 General Parallel Algorithm Design for NP-Complete Problems*

The previous sections have described generic search techniques, parallel architectures, and parallel algorithm design methods. This section combines, at a high level, the information presented to develop a general structure for searching NP-complete problems on parallel architectures. Most algorithms for NP-complete search problems incorporate some form of a branch-and-bound or divide-and-conquer algorithm. Both of these algorithms are symmetrically appealing from a parallel programming approach since they map

directly to a tree structure; therefore, the use of these two search techniques in a parallel algorithm is presented. But first, common questions must be asked before designing a parallel program using either algorithm. For instance (23, 46):

- How should the data be distributed across the processors and what should the grain size be?
- If the architecture is loosely coupled (e.g., a cube-connected architecture), should a global best cost be maintained and transmitted to the searching processors? How often and in what manner should the transfer be accomplished? For example, should all processors be notified immediately by a central processor whenever a better cost is discovered or should the searching processors poll each other for a better cost? These questions arise since frequent transmission of a better cost may create communications bottlenecks, but global knowledge of the best cost is necessary to prevent duplication of subbranches in individual processors.
- Is it profitable to compute a near optimal solution using, perhaps, a greedy method before the parallel search begins? If so, this first solution could be used to bound the initial search.
- Should static or dynamic load balancing be used? Dynamic load balancing could be computationally expensive; therefore, is it possible to predetermine an acceptable static load balance? And if dynamic load balancing is implemented, how does the algorithm detect termination of the search?

The answers to all of these questions are obviously problem dependent and greatly influence the design. Furthermore, the answers to all but the first question can not be fully answered without implementing an algorithm and then observing its performance; therefore, the answers to these questions are left to future chapters. The next section focuses on the parallelization of a divide-and-conquer algorithm. In distributed systems, divide-and-conquer is typically used to distribute the data for processing by the individual processors.

*2.6.1 Divide-and-Conquer* "Parallel processing of divide-and-conquer algorithms can be classified into three types:

1. Multiprocessors are connected in the form of a tree, especially a binary tree.
2. Virtual tree machine consisting of a number of processors with private memory.
3. All processors are connected to a common memory through a common bus" (56:95).

Recall that a divide-and-conquer search occurs in three phases: startup, computation, and wind-down. During startup, the problem is divided into smaller subproblems which are then divided among the processors. The computation phase occurs when each processor searches its subproblem(s). The wind-down phase occurs as each processor completes its search and transmits its results (56:95). The time spent on startup is usually insignificant compared to computation and wind-down. Furthermore, the computation and wind-down phases for NP-complete problems are unpredictable since NP-complete search problems are typically inhomogeneous . Therefore, the amount of searching each processor must accomplish can be radically different and some method of dynamic load balancing is usually required to achieve maximum performance (25:8).

For the search of NP-complete problems, a divide-and-conquer algorithm partitions the search tree among the cube-connected processors. The individual processors search their subbranch and report the results to a central processor. Further division of remaining search trees is accomplished and the finished processor is given another subbranch of the search tree. Divide-and-conquer is used to distribute subbranches to the searching processors which then use a branch-and-bound algorithm to search the subbranch.

*2.6.2 Branch-and-Bound* In many parallel search algorithms, branch-and-bound algorithms are used to search subproblems created by divide-and-conquer algorithms (53). A branch-and-bound algorithm consists of the same three phases as the divide-and-conquer. During the startup phase, disjoint subproblems are generated and then assigned to each processor either dynamically (as a processor finishes with one subproblem it receives another) or statically using some heuristic function to determine which processor receives which subproblem(s). The most significant problem associated with the generation of the

subproblems is the generation of subproblems which do not contain the "best lower bounds in the global sense" (1:1497). While the generation of less than optimal subproblems does not affect the correctness of the algorithm, "it does increase the number of subproblems examined by the algorithm" (1:1497) and may increase the time required to solve the problem.

Each processor searches an assigned subproblem during the computation phase. Especially important during this phase is the early broadcast of the global best solution. As each processor computes a solution, that solution is compared against the best solution obtained thus far by all processors (global best solution). The global best is used along with the local best solution in the branch-and-bound elimination procedure to prune the search tree (53:1528). Two problems may arise from the broadcast of the global best solution. First, it may cause a load imbalance in the parallel search. For example, the global best may allow one processor to prune nearly its entire search tree while the search tree of another processor is only partially pruned. Such an imbalance increases the likelihood of processor idle time and thus lowers the efficiency of the search. The second problem arising from a broadcast of the global best solution results when the global best is not transmitted early enough. In this case, processors may duplicate portions of the search (53:1526).

Division of the search tree and broadcast of a global best solution results in the search tree being searched in a different order in the parallel and serial algorithms. This can result in an anomaly discussed by Lai and Sahni (40) and summed up by Wah, Li, and Yu (56) as follows:

> A $k$-fold speedup is expected when $k$ processors are used. However, simulations have shown that the number of iterations for a parallel branch-and-bound algorithm using $k$ processors can be more than the number of iterations of the best serial algorithm (this phenomenon is a detrimental anomaly); less than one-$k^{th}$ of the number of iterations of the best serial algorithm (an acceleration anomaly); or less than the number of iterations of the best serial algorithm, but more than one-$k^{th}$ of the number of iterations of the best serial algorithm (an deceleration anomaly). (56:95)

The wind-down phase occurs when the number of subproblems is less than the number of processors. Increasingly more processors become idle until the search is complete.

In most instances, the ideal situation is for all processors to work until the solution is obtained (i.e., no processor is ever idle). The inhomogeneous nature of an NP-complete problem's search space makes this difficult to accomplish.

In their most basic form, parallel divide-and-conquer and branch-and-bound algorithms are multiple versions of serial algorithms executing on serial processors searching a subset of the total search space with the search bounded by the global and local best solutions. The previous sections examined parallel divide-and-conquer and branch-and-bound techniques. This research draws on both techniques to design a parallel search technique for NP-complete problems.

### *2.7 Summary*

This chapter has focused on the topics of NP-completeness, the SCP, generic search techniques, parallel architectures, and parallel algorithm design for NP-complete problems. To be NP-complete a problem must satisfy two requirements: (1) it must be in the class $\mathcal{NP}$, and (2) it must be transformable to all other NP-complete problems in polynomial time and vice-versa.

The SCP is briefly explained to be the selection of a set of columns from a 0–1 matrix such that the cost of the covering columns is minimal and all rows of the matrix are covered by at least one element in any of the covering columns. A more detailed explanation of the SCP is provided in Section E.6. Included are the formal definition, search techniques including the construction of a *table* to assist in the search, a priori reductions, a dominance test, and a lower bound test.

The search techniques presented in this chapter are generic and constitute a base from which many common search algorithms are developed. Since the objective of this research is the optimal solution of NP-complete problems, only optimal search techniques are used; hence, the greedy method and probabilistic search methods are not considered beyond their introduction in this chapter. A search based on divide-and-conquer and branch-and-bound techniques offers the most promise for an efficient parallel search.

The parallel architecture section briefly discusses common methods for classifying

parallel architectures and introduces current parallel architectures. References are given to more complete descriptions. The cube-connected parallel architecture is discussed since the algorithms for this research are implemented on a cube-connected computer. The cube-connected computer is especially suited to a tree search since it's network is easily configured into a tree structure. The last topic covered in the parallel architecture section is the design of parallel software. Four similar approaches are presented followed by a discussion of UNITY. UNITY is a structured design syntax aimed at incorporating a formal notation into the parallel software design process. UNITY versions of the SCP are developed in Chapters III and IV.

The application of parallel program design to NP-complete problems is presented. Application of parallel divide-and-conquer and parallel branch-and-bound search techniques is discussed and the algorithms for this research are based on both of these search techniques. More specifically, the subbranches are allocated to the processors using divide-and-conquer and searched with a branch-and-bound algorithm.

Though somewhat lengthy, this survey only introduces the available information. An entire library can be constructed using just the articles and books covering search techniques and parallel processing. The ultimate objective of this review is to sufficiently cover the topics so that the reader can better appreciate the decisions made during the design of the parallel SCP algorithms conducted in next two chapters.

# *III. Methodology & Design*

## *3.1 Introduction*

The purpose of this chapter is to discuss the research methodology and conduct a preliminary design of a parallel version of the set covering problem (SCP) for the Intel iPSC/2 Concurrent Supercomputer. The methodology is contained in Section 3.2 and the preliminary design in Section 3.3. The preliminary design defines the control and data structures for all the SCP routines, develops UNITY metaprograms for a serial and parallel branch-and-bound SCP algorithm, and describes the parallel reductions. The detailed design of the parallel SCP, reduction techniques, dominance test, and lower bound test are described in Chapter IV. Section 3.4 is a complexity analysis of the SCP and Section 3.5 discusses the computer equipment (iPSC/2 hardware and software) to be used in the implementation of this design.

## *3.2 Research Methodology*

The following methodology outlines a systematic approach for implementing a parallel program to solve the general SCP. The problem is addressed in the following manner:

1. Develop a conceptual and in-depth understanding of the SCP in terms of its overall structure. In other words, define the nature of the problem.
2. Survey and evaluate current serial SCP algorithms and select the "best" candidates for further development.
3. Develop parallel algorithms for solving the SCP and evaluate their expected performance using complexity analysis.
4. Implement (program) the best parallel algorithm on the iPSC/2.
5. Evaluate the performance of the parallel program or programs using performance metrics developed in the course of this research.

The following paragraphs summarize the various parts of the research methodology:

First, a thorough understanding of the SCP must be developed before attempting to construct any algorithms. Conceptually, the SCP is relatively easy to understand: it is a search of a 0–1 matrix to find the minimum cost covering sets (Sections 2.3 and E.6). The methods for conducting such a search are numerous and are presented in the previous chapter. In addition to the basic search, various techniques are available to reduce the input problem's dimensions and to improve the efficiency of the search. Many articles relating to the SCP's solution and applications were obtained and studied (18, 7, 52, 27, 43, 17).

The next step is to conduct a survey of and evaluate current SCP algorithms. This survey is necessary to prevent any duplication of previous work. Particular effort is concentrated on obtaining parallel SCP algorithms or algorithms for other NP-complete problems that are applicable to the SCP. The results of the survey are summarized in Chapter II and indicate that only serial SCP algorithms have been published (17, 18, 7, 52, 27, 43); however, since all NP-complete problems are related and several references were obtained on parallel implementations of other NP-complete problems (56, 21, 23, 46, 53, 40, 1, 50, 25), it is reasonable to assume that the references can offer suggestions that apply to this project. Information of interest is the application of general, parallel search algorithms to NP-complete problems, data structures used in the parallel solution of NP-complete problems, and methods for load balancing parallel algorithms.

The serial SCP algorithm obtained from the survey is evaluated using the parallel algorithm characteristics presented in Section 2.5.3. The characteristics of interest are the problem's inherent parallelism, its load balancing requirements (hardware dependent), and its mapping to a parallel architecture. Christofides' (17) serial SCP algorithm, Section 2.3, is modified for this research so that it can effectively search for a set cover in a parallel environment. Such modifications include the distribution of data and control as well as the load balancing required to improve the efficiency of the parallel search algorithm.

The third step is to develop the parallel algorithms and analyze their expected performance. A design syntax called UNITY, mentioned in Section 2.5.3 and described in SectionIII:UNITY, is iteratively applied to develop the parallel algorithms. Unlike algorithms for serial machines, parallel algorithms are generally developed for specific parallel architectures. Such factors as the number of processors, the computer's interconnection

network, and the processor's access to memory must be considered in the algorithm design and analysis at some level of development (reference Section 2.5.3). Furthermore, good engineering practice requires a complexity analysis be performed on the algorithms to determine their expected performance. This complexity analysis lends insight into the algorithms and, in many cases, can indicate a less than efficient algorithm design. Therefore, a worst and best case complexity analysis of the SCP algorithms is conducted. The worst case analysis assumes a serial computer architecture since the worst case must occur when only one processor is searching. The best case analysis assumes the SCP search is executed on a computer specifically designed for the associated search algorithms. This mythical computer has an unlimited number of processors, an interconnection network specifically designed for the problem's architecture, unconstrained access to memory, and no communications overhead.

Following the design of the parallel algorithms, the forth step is to implement the algorithms on the iPSC/2. As mentioned in Section 1.5, much of the initial software development is efficiently done on a personal computer using Turbo C. The library routines are defined a little differently between Turbo C and the iPSC/2's C compiler. For instance, Turbo C's `time` routine returns a different type than the iPSC/2's C compiler. Hence, the programs written for Turbo C must be modified when hosted on the iPSC/2. C's use of conditional compilation makes this conversion simple. A more detailed description of both Turbo C and the iPSC/2's C compiler are provided in Section 3.5.

The final step in this research methodology is to empirically analyze the program's performance in terms of its efficiency, effectiveness, time, and space requirements. The analysis consists of executing the program with selected input data and collecting various performance metrics. Clearly, the following list of metrics is not all encompassing; however, it is sufficient for the purpose of this research:

**Total Program Execution Time** — Total execution time for the SCP program.

**Search Time** — Time the algorithms actually spent searching. Does not include time to build the table, sort the data, and so forth.

**Expanded Nodes** — Number of nodes expanded in the search tree by each searching processor.

**Processor Idle Time** — Time the searching processors are waiting for data to search.

**Minimum Solution Time** — Program execution time until the optimal solution was first found.

**Global Best Cost Broadcasts** — Number of times the global best cost is broadcast to the searching processors.

**Support Time** — Time to sort the input data, build the table, and execute any reductions.

**Search Efficiency** — Sum of the individual processor search times divided by the product of the total program execution time and the number of searching processors.

**Speedup** — Execution time of the best serial algorithm divided by the execution time of the parallel algorithm.

**Load Balance Time** — Time each processor spent load balancing instead of searching.

**Search State** — The state of the search at each node in the search tree.

An evaluation of the run-time analysis probably necessitates slight program changes to eliminate program bottlenecks. Such changes are expected to be minor given the thorough algorithm design and complexity analysis previously accomplished. Testing consists of verifying the efficiency and effectiveness of the algorithms by careful selection of a set of test files as described further in Chapter V.

The preceding methodology is necessary to guide the development and analysis of a parallel algorithm to solve NP-complete problems. Other approaches to solving this problem are certainly available (56, 4, 14, 38, 19, 50, 25, 16, 6, 15, 28) and three were previously presented in Section 2.5.3. The complete design of the parallel programs involves a preliminary and detailed design. Although the search process is relatively easy to understand, an efficient and effective parallel algorithm is not easy to design for reasons discussed in Chapter II. The following sections in this chapter summarize the preliminary design whereas Chapter IV presents the detailed design.

### *3.3 SCP Program Design*

The design of a parallel program involves many tradeoffs. The choices for dividing the control and data structures are numerous and the designer can quickly become confused by the myriad of choices. Given the relative infancy of parallel program design, it is difficult to prove any one method of decomposition is more efficient than the multitude of others. In fact, it becomes difficult to justify the designer's decision of just what *efficient* means. Furthermore, as with any project, there is always a tradeoff between time spent designing versus time allotted to the project as the chosen control and data structures must ultimately be programmed. The control/data structures must not be so complex that they can not be implemented within the available development time.

A basic, branch-and-bound, serial SCP algorithm is presented in Christofides (17) and forms the basis for the parallel algorithms developed in this project. Although his algorithm is written and implemented on a serial machine, many of the steps involved in finding a cover are applicable to parallel machines. As stated in Chapter II, an optimal search may involve a considerable amount of bookkeeping; therefore, Christofides' algorithm preprocesses the data by constructing a table. The table eases the combining of columns by ensuring a cover is present whenever one set is chosen from each block in the table. In addition to the branch-and-bound search, Christofides outlines several algorithms which potentially decrease the amount of searching required (e.g., reductions, dominance testing, and a lower bound test). The following high-level algorithm provides a starting point for the design:

**Algorithm** *High-level Search*
    Reduce matrix
    Build 0–1 table
    Search table for an optimal set cover
**End** *High-level Search*

The routines to build and search the table are the core of the SCP in both the serial and parallel algorithms; hence, these routines are designed first. The reductions, dominance test, and lower bound test routines explained in Section E.6 are added to the design later. The designer must first divide the problem control and data structures such that all processors share in the search process (i.e., no idle processors). This division

or parallelization of the problem usually results in control and data structures which are classified as either functional or data parallel. Functional parallelism divides the problem into a set of individual processes which are scheduled and executed based on precedence relations. Data parallelism replicates an inherently sequential algorithm on all processors and then partitions or divides the data between the processors. The processors then execute their algorithm and combine the partial solutions into a single answer (6, 19).

Simulations and operating systems are good examples of functional parallelism (19). These problems can be divided into essentially autonomous processes with the relationship between the processes represented by a precedence data structure graph. Processes are scheduled on processors and communicate as necessary to ensure the precedence relationships are satisfied. The search graph (control structure) generated by the search for an optimal set cover also defines a precedence relationship. At each stage of the search down a branch of the graph, a required set of nodes must be expanded first. Although possible to structure the SCP in this manner, the number of precedence relationships grow exponentially making it difficult to manage (ref. Section 3.4). It is thus more prudent to structure the SCP as a data parallelism problem.

A data parallelism approach to parallelizing the SCP divides the search tree among the processors in such a manner that each processor searches a different subbranch of the search tree! These processors then execute an identical search algorithm on their allotted portion of the search tree. When a processor finishes its search, it sends its best solution to a central, controlling processor. The controlling processor retains the best solution obtained by any processor. Once all processors have completed their search, the controlling processor contains the optimal cover for the original input problem. Since autonomous processors are searching subbranches of the search tree, two important questions must be considered:

- Should the best cost maintained by a controlling processor be transmitted to the searching processors for inclusion in their bounding functions? As each processor searches its allotted subbranch, it retains its best cover for future pruning of its search tree. If the searching processors were to transmit their best cover at some suitable stage of their search and receive a copy of the best cost maintained by the

controlling processor (i.e., the best cost found so far by all searching processors), it could prune its search tree even further.

- How should the subbranches be developed and how should they be divided among the searching processors? The central controlling processor could determine the allocation of subbranches once at the beginning of the search (coarse grain approach with static load balancing) or it could determine the allocation as the processors are searching (dynamic load balancing). An algorithm based on a coarse grain approach is more likely to have idle processors because NP-complete problems are inhomogeneous making it difficult, if not impossible, to allocate subbranches requiring equal expansion. On the other hand, a dynamic load balanced approach dynamically develops and allocates the subbranches and may not induce as much processor idle-time. The major drawback with dynamic load balancing is of course the added complexity of the algorithms and the overhead time required to divide and allocate the subbranches.

*3.3.1 Control/Data Structures* A functional parallelism algorithm is initially appealing since it maps the algorithm's control structure directly to the computer architecture. However, the number of precedence relationships can grow exponentially for NP-complete problems making the bookkeeping difficult. Therefore, a data parallelism approach is used to implement the SCP. This method of parallel data decomposition leads to a quicker design and implementation on a project that promises to be quite large and software intensive. Furthermore, many of the articles reviewed in conjunction with this research topic have implemented NP-complete searches using similar techniques for parallel decomposition (1, 23, 46, 53). Initially, the data is allocated using a coarse grain approach with static allocation of the search tree; in other words, the input data is partitioned into large subbranches with each searching processor receiving at least one subbranch. This method of data decomposition is straight forward and requires development of all the major program algorithms. The static allocation method is then analyzed to determine if a dynamic allocation of the initial search tree is necessary. The key performance parameter to be observed is individual processor idle-time indicating the need for further load balanc-

ing. Other performance metrics observed are the number of nodes expanded by individual processors and the number of times the global best cost maintained by the controlling processor is transmitted to the searching processors.

Given the decision to develop the SCP algorithms using a data parallelism approach, the input matrix is broadcast to all processors participating in the search. Each processor constructs a table (Section E.6.5, page E-12) to assist in the search (17, 18). The processors then expand the search tree using the table and a breadth-first algorithm (explained on page 3-13) until each processor has at least one distinct subbranch of the search tree. The processors then search their allotted subbranch. During the search process, the searching processors transmit their best cover to the controlling processor whenever their best cost is better than their copy of the global best cost. Each searching processor also checks its receive buffer for a new global best cost after each backtracking step and updates its copy of the global best cost if the received global best cost is better than the locally maintained copy of the global best cost.

Now that a high-level, preliminary design of a parallel algorithm for the SCP exists, the data structures required to implement this algorithm must be defined. The 0–1 matrix is a two dimensional matrix containing a '1' in every location where a row is covered by a column and a '0' elsewhere. The columns of the matrix have an associated cost; hence, the costs are stored in a cost vector (one-dimensional matrix). Each processor constructs a table to assist in the search. The data structure for the table could be another two-dimensional matrix explicitly listing the rows and columns in the matrix. This approach builds the table illustrated in Figure E.11 on page E-12. The size of the resulting matrix is potentially $nrows^2 \times ncols$ where *nrows* is the number of rows in the matrix and *ncols* is the number of columns. In addition to the table, vectors are required to keep track of which sets (columns) are currently in the list of covering sets and which rows are currently covered.

At this stage of the design, the data structures just defined could be used to search for a set cover. However, it is possible to construct the table as a linked-list of pointers into the 0–1 matrix and decrease memory usage. A linked-list table complicates the search algorithm but the savings in memory might be needed by the search algorithm for stor-

age of the stack, dominance test $L$ matrix, or the lower bound test $D$ and $D'$ matrices (Sections E.6.6 and E.6.7).

One final modification to the table is suggested by Christofides (17:42). He suggests arranging the blocks of the table in ascending order according to the number of columns in each block. In other words, sort the rows in increasing order according to the number of 1's in each row. The resulting table is shown in Figure 3.1 and a search of this table is shown in Figure 3.2. Sorting the rows in this manner decreases, in many instances, the number of nodes expanded since the search tree traversal is more vertical than horizontal. A decrease in the number of expanded nodes corresponds to a decrease in the solution time. For example, consider the table constructed in Section E.11 and the resulting search tree shown in Figure E.12. A comparison between the two search trees clearly illustrates the vertical versus horizontal expansion. Obviously, there exist problems which violate this assumption, but such is the nature of NP-complete problems.

Figure 3.3 illustrates the linked-list table (hence forth known as the table) used in this implementation. The rows and columns in the table are indices into a vertex vector, Figure 3.4, and a set vector, Figure 3.5. Each vertex record contains an index to a row

| | | Blocks | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | 0 | 1 | | | 2 | | | 3 | | | | 4 | | | | 5 | | | | |
| | | Columns | | | | | | | | | | | | | | | | | | | |
| | | 3 | 0 | 6 | 1 | 5 | 4 | 6 | 4 | 7 | 6 | 1 | 5 | 0 | 2 | 7 | 5 | 0 | 2 | 7 | 1 |
| | 2 | 1 | | | | | | | | | | | | | | | | | | | |
| | 5 | 0 | 1 | 1 | 1 | | | | | | | | | | | | | | | | |
| Rows | 4 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | | | | | | | | | | | | | |
| | 3 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | | | | | | | | | |
| | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | | | | | |
| | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| | | 8 | 4 | 6 | 7 | 2 | 3 | 6 | 3 | 5 | 6 | 7 | 2 | 4 | 5 | 5 | 2 | 4 | 5 | 5 | 7 |

Costs

Figure 3.1. New Table for Figure 2.1

in the matrix (**Index**), the number of 1's in the indexed row (**Cardinality**), and whether the indexed row is covered (**Covered**). Each set record contains an index to a column in

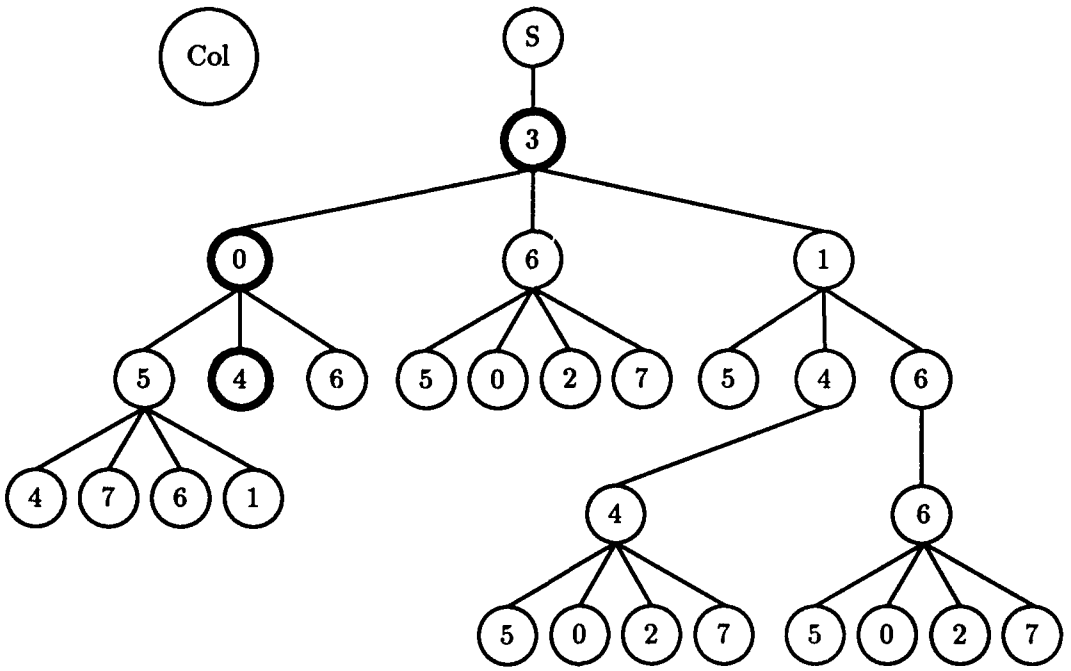



Figure 3.2. Table Data Structure

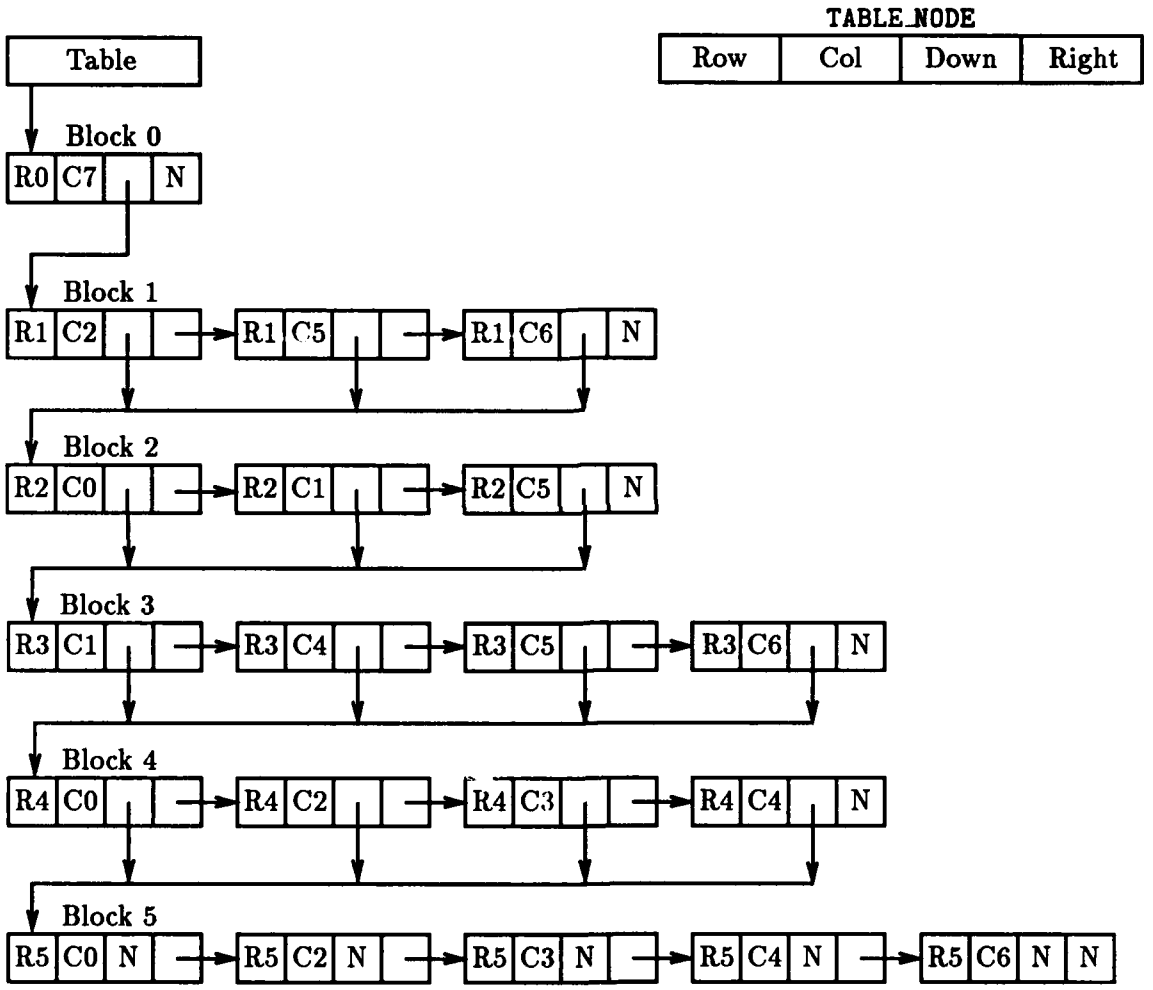



Figure 3.3. Table Data Structure

the matrix (**Index**), the cost associated with the indexed column (**Cost**), and whether the indexed column has been used as a covering set (**Used**). A vertex/set vector consists of a list of vertex/set records as shown in Figures 3.4 and 3.5.

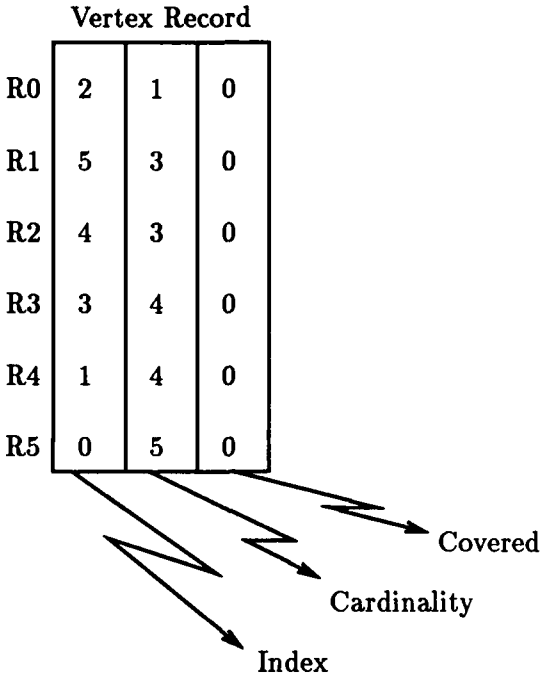



Figure 3.4. Vertex Record

Set Record

| C0 | C1 | C2 | C3 | C4 | C5 | C6 | C7 | |
|---|---|---|---|---|---|---|---|---|
| 5 | 4 | 0 | 2 | 7 | 6 | 1 | 3 | Index |
| 2 | 3 | 4 | 5 | 5 | 6 | 7 | 8 | Cost |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | Used |

Figure 3.5. Set Record

Realize, of course, that a second level of indirection is now required to access the 0–1 matrix. The algorithm is more complicated, but the 0–1 matrix is never changed and operations such as sorting are performed more efficiently on the vertex and set records than could be performed on the entire 0–1 matrix. For example, consider sorting the columns

of an $N \times N$ matrix according to ascending column cost. The matrix consists of $N^2$ elements and the set vector consists of $N$ records. To sort the column costs is $O(N \log N)$ regardless of whether the matrix elements or set vector records are moved (10:462). Now, consider the number of memory moves required to sort the matrix versus the number of moves required to sort the set vector. To sort the matrix requires $N$ memory swaps for each of the $O(N \log N)$ comparisons for an order-of $O(N^2 \log N)$. On the other hand, the set vector is composed of only three elements and a sort of it is $O(3N \log N)$. Hence, provided $N > 3$, it is much more efficient to sort the set vector rather than the matrix elements. A similar argument can be made for sorting the vertex vector rather than the matrix elements.

Now that the basic data structures exist for the parallel SCP solution, the next task is to develop an algorithm to divide the search tree among the searching processors. A breadth-first expansion, as stated, divides the search tree so that each processor is assigned a distinct section of the initial search tree. The expansion is accomplished with the aid of the table. For example, consider a breadth-first expansion of Blocks 0 and 1 in Figure 3.3. An expansion results in three separate tables which can be assigned to separate processors for searching. Rather than construct three separate tables, it is more economical (saves time and space) to simply save lists of expanded **TABLE_NODEs** and let them continue to point into the full table, Figure 3.6. The searching processors are then instructed to search their copy of the full table starting with a specified list of expanded **TABLE_NODEs**.

The algorithm for accomplishing the expansion uses a queue to keep the lists of expanded **TABLE_NODEs** and continually extracts an expanded list off the front of the queue and appends a newly expanded list onto the rear of the queue.

**Algorithm** *Breadth-first Expansion (SCP)*
    Put all **TABLE_NODEs** from Block 0 in the queue
    Loop until every searcher has at least one subtree to search
        Point to the first **TABLE_NODE** in the next block
        Extract a list off the queue
        Loop for all **TABLE_NODEs** in this block
            Append **TABLE_NODE** to list
            Insert new list into the queue
            Point to next **TABLE_NODE** in this block
        End loop
    End loop
**End** *Breadth-first Expansion (SCP)*

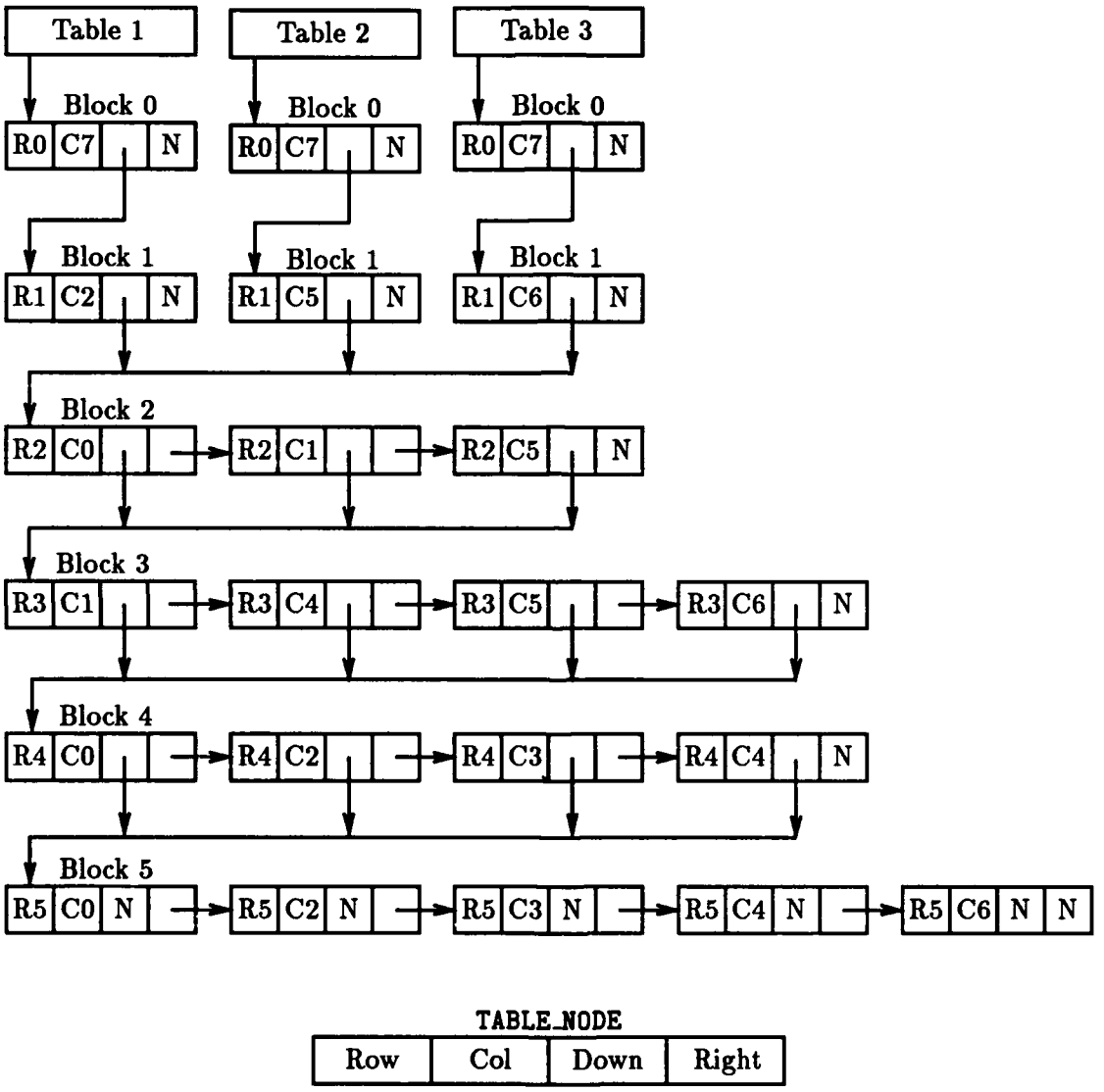



Figure 3.6. Three Subtables for Three Searching Processors

The expansion illustrated in Figure 3.6 partitions the search tree for a maximum of three searching processors. If additional searchers are present, the breadth-first expansion continues to expand the table until enough subtables (i.e., subtrees) are available such that all searchers receive an expanded list of **TABLE_NODEs**.

A couple of subtle observations regarding the search graph are worth noting. First, the breadth-first expansion may expand the entire search tree if the tree is small and many searching processors are available. In effect, a breadth-first search of the entire search tree occurs and, as stated in Section 2.4.2.2, such a search is inefficient. The second observation relates to reduction #2 described in Section E.6.4. Recall that this reduction checks the input 0–1 matrix for rows covered by only one column. Since the rows of the table are now sorted in ascending order, any rows covered by only one column are listed first in the table and are immediately added to the list of covering sets where they remain until the search is complete. In effect, sorting the rows in ascending order implements reduction #2 and potentially improves the efficiency of the search at no additional cost in execution time.

The table is a linked-list of pointers which point to the vertex and set records via an index into these records. Likewise, the vertex and set records contain indices into the 0–1 matrix. Only one copy of the 0–1 matrix is maintained in memory and is accessed via the table, vertex record, and set record. The vertex record is sorted in ascending order according to the row cardinality and the set record is sorted in ascending order according to the cost of the columns.

Construction of the table is now complete and an initial algorithm exists to partition the table between any number of searching processors. Next step is to develop an algorithm to search the table for an optimal cover.

*3.3.2 Search Algorithm* As previously stated in Section 2.5.3, a design syntax language known as UNITY is used to design the search algorithms. The design process is to develop increasingly more complex UNITY representations (i.e., metaprograms) until the UNITY program is sufficiently developed to map to a target architecture. Before the UNITY programs are presented, the notation is explained in the following section.

*3.3.2.1 UNITY Notation* An explanation of UNITY is contained in Chandy and Misra (16) from which this brief background is extracted.

> Given a specification and a target architecture, a programmer's task is to derive a program with its proof, select a mapping that maps programs to the given target architecture, and then evaluate complexity measures. A program consists of a declaration of variables, a specification of their initial values, and a set of multiple-assignment statements. A program execution starts from any state satisfying the initial condition and goes on forever; in each step of execution some assignment statement is selected nondeterministically and executed. Nondeterministic selection is constrained by the following "fairness" rule: Every statement is selected infinitely often. (16:8–9)

A UNITY program consists of the following four sections:

**declare** — The declare-section names the variables used in the program and their types.

**always** — The always-section defines certain variables as functions of others.

**initially** — The initially-section is used to define initial values of some variables; uninitialized variables have arbitrary initial values. An initialization is denoted by =.

**assign** — The assign-section contains a set of of assignment statements denoted by :=.

An assignment statement may consist of multiple assignments

$$x, y, z := 0, 1, 2$$

or as a set of assignment components separated by ||.

$$x := 0 \;||\; y := 1 \;||\; z := 3$$

In either case, all variables are assigned simultaneously.

The last structure of importance is a case statement and is denoted as follows:

$$\begin{array}{lll} x := & -1 & \text{if } y < 0 \sim \\ & 0 & \text{if } y = 0 \sim \\ & 1 & \text{if } y > 0 \end{array}$$

in which $x$ is assigned a value depending on the evaluation of $y$.

The above notation should provide enough background to interpret the UNITY programs presented in the following sections. As before, more information is available in Chapter 2 of Chandy and Misra (16).

*3.3.2.2 Serial SCP UNITY Program* Since the parallel SCP programs are derived in part from serial SCP programs, a serial SCP UNITY program is developed first and then transformed to a parallel UNITY program. The serial SCP algorithm is derived from Christofides (17:41–42) branch-and-bound SCP algorithm. Recall from Section 2.4.6 that a branch-and-bound search executes a depth-first expansion with bounding functions which limit the depth of the search graph. Four such bounding functions are defined for the SCP and are based on the cost of the current set of covering sets, the existence of a cover, the result from a dominance test, and the result from a lower bound test.

For example, consider the search for the optimal set cover of the 0–1 matrix in Figure 2.1. For the purpose of this example, the bounding functions are limited to the cost of the current set of covering sets and the existence of a cover. The reader should consult Section E.6 for a more detailed explanation of the search process involving all four bounding functions. The search algorithm selects columns from the matrix according to the following two rules: 1) the columns are chosen from left to right in the matrix and 2) a candidate column must cover a row not covered by the current set of columns. The search algorithm backtracks whenever all the rows are covered or the cumulative cost of the columns exceeds a previous cover. The resulting search graph is illustrated in Figure 3.7. The search graph represents the process of adding and removing columns for a list of covering sets. At the leaf node in each path, the search algorithm backtracks for one of three reasons: 1) a better cover has been found, 2) inclusion of the next column exceeds the best cost cover thus far, or 3) no solution is possible down the current path. The large nodes in the search graph are the columns which are added to the list of covering sets. The small nodes represent columns which were considered for inclusion but were not included because to do so would result in a cost greater than the best cost obtain thus far or no solution is possible down the current path. The cost of each new cover is denoted at the leaf nodes as $\hat{z}$. The reader is referred to Appendix E for a detailed explanation of the serial SCP search process.

The following paragraphs first explain the four bounding functions and then present a UNITY program for a serial branch-and-bound SCP based on the *Subset-Sum* UNITY program presented in Chandy and Misra (16:444–446).

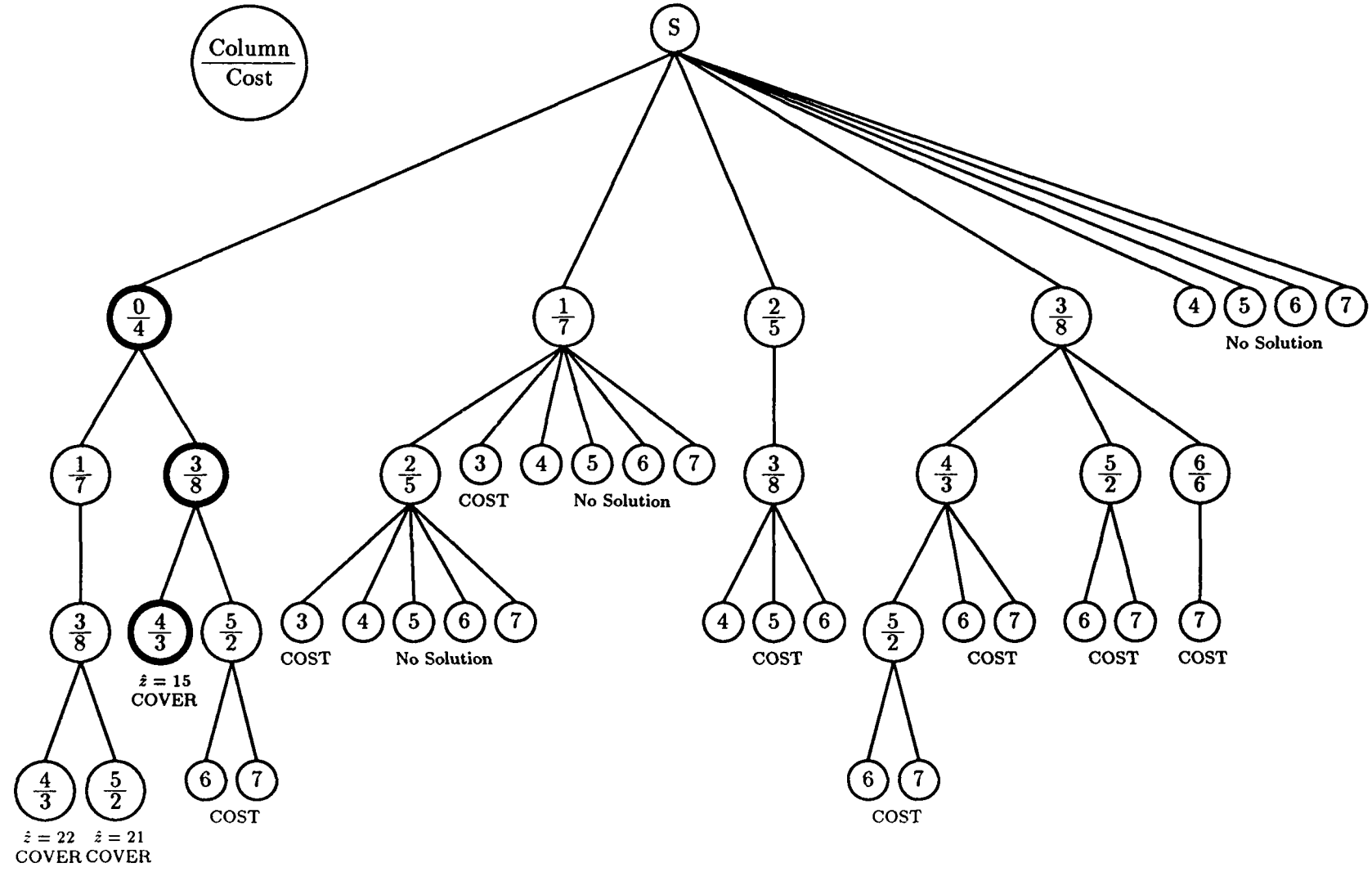



Figure 3.7. SCP Search of Figure 2.1

**Cost:** The cost of a cover is the cumulative cost of the sets chosen to cover the vertices. Let $\hat{z}$ represent the best cost obtained for a cover and $z$ represent the cost of the current state of the search comprised of a set of covering sets. If $z \geq \hat{z}$ the algorithm backtracks since a better solution can not be found following the current path.

**Cover:** A set cover was defined in equation E.1 on page E-4. Let *cover* be a boolean variable that is true if the current list of covering sets contains a cover per equation E.1.

$$cover = \text{TRUE} \quad \text{if} \quad \bigcup_{i=1}^{k} S_{ji} = R \tag{3.1}$$

If *cover* is true, the algorithm backtracks since the addition of more covering sets only serves to increase the cost of the cover.

**Dominance test:** Let $D$ be a boolean variable that is true if the current state of the search is dominated by a previously saved state. Let $E$ represent the current state of the search with a cost $Z_E$ and $E_p$ represent previously saved states ($E$'s) with a cost of $Z_p$.

$$D = \text{TRUE} \quad \text{if} \quad E \subseteq E_p \wedge Z_E \geq Z_p \tag{3.2}$$

Section E.6.6 on page E-13 explains the details of the dominance test. If $D$ is true, the algorithm backtracks since the current set of covered vertices is dominated by a previously saved set of covered vertices. A better solution, by construction, can not be found down a dominated path.

**Lower-bound test:** Let $L$ be a boolean variable that is true if the lowest possible cost down the current path exceeds the best cost. Let $l$ represent the lowest possible cost (achievable or not) down the current path and let $\hat{z}$ represent the best cost obtained thus far.

$$L = \text{TRUE} \quad \text{if} \quad l \geq \hat{z} \tag{3.3}$$

Section E.6.7 on page E-17 explains the details of the lower bound test. If $L$ is true, the algorithm backtracks since the current path can not lead to a better solution.

In the following UNITY program, *seq* is a list of covering sets representing the current state of the search sorted in lexicographical order with a cost of $z$ and $\hat{S}$ is the optimal cover with a cost of $\hat{z}$. Furthermore, for ease of programming, let there exist an artificial covering set $S[n]$ that covers all vertices and has a cost of $\sum_{i=0}^{n-1} C[i] + 1$. Hence, there are

$S[0 \cdots n]$ covering sets that must contain at least one cover.

**Program** *Serial SCP*

**declare**

| | | |
|---|---|---|
| $C[v]$ | : int | {cost of covering set $v$} |
| $l$ | : int | {calculated lower bound} |
| $v$ | : int | {index variable} |
| $z$ | : int | {cost of the current state of the search} |
| $\hat{z}$ | : int | {cost of the best cover} |
| *cover* | : boolean | {TRUE if a cover exists} |
| $D$ | : boolean | {TRUE if a dominate set exists} |
| $L$ | : boolean | {TRUE if lower bound is exceeded} |
| *seq* | : set | {current state of the search} |
| $\hat{S}$ | : set | {the best cover} |

**always**

$1 \leq v \leq n$

$z = \langle +i : i \in seq :: C[i] \rangle$

$\hat{z} = z$ if $cover \wedge z < \hat{z} \sim$

$\hat{z}$

$\hat{S} = seq$ if $cover \wedge z < \hat{z} \sim$

$\hat{S}$

$D(v) =$ TRUE if $seq \cup v$ is dominated by a previous $seq \sim$

FALSE

$L(v) =$ TRUE if $z + C[v] \geq l \sim$

FALSE

**initially**

$l, v, z, \hat{z} = 0, 0, 0, \infty$

$seq, \hat{S} = null, null$

**assign**

| | | | |
|---|---|---|---|
| $seq,$ | $v$ | := | |
| $seq; v,$ | $v+1$ | if | $z + C[v] < \hat{z} \wedge \overline{cover} \wedge \overline{D(seq; v)} \wedge \overline{L(seq; v)} \sim$ |
| $\text{pop}(seq),$ | $\text{top}(seq)+1$ | if | $seq \neq null \wedge (z + C[v] \geq \hat{z} \vee cover \vee D(seq; v) \vee L(seq; v)) \sim$ |
| $null,$ | $n$ | if | $seq = null \wedge (z + C[v] \geq \hat{z} \vee cover \vee D(seq; v) \vee L(seq; v))$ |

**End** {*Serial SCP*}

The always-section performs several functions:

1. It ensures that the cost ($z$) of the current covering sets ($seq$) is the sum of the costs of the individual covering sets ($C[v]$).
2. It keeps the best cost ($\hat{z}$) current by setting it to a lower cost cover if one is found. In addition, it keeps the best cover ($\hat{S}$) current by setting it to the lowest cost cover.
3. It keeps the bounding functions, $D$ and $L$, current.

There are three possible assignments in the assign-section. The first assignment extends the set of covering sets ($seq$) by adding the next covering set, indexed by $v$, to $seq$ and incrementing the index to the next covering set. The second assignment is the backtrack step. The backtrack removes the last covering set (pop($seq$)) from $seq$ and sets the index to the covering set following the removed covering set. The last assignment ensures that the search process does not repeat or abort.

Now that the program has been developed, an invariant, fixed point, and progress condition are derived to ensure that the UNITY program correctly finds the optimal solution. Upon completion of the program, a set cover has been found if:

$$\hat{S} \neq S[n] \tag{3.4}$$

The following invariant states that the sequence $(seq; v)$ is constantly increasing; thus, all possible covers are investigated using covering sets $S[0 \cdots n]$. The second conjunct states that a cover is eventually found. This must be true since an artificial cover ($S[n]$) was included in the list of covering sets. However, if $\hat{S} = S[n]$, a cover involving the real covering sets was not found.

**invariant**

$(seq; v)$ is an increasing sequence of covering sets
$\wedge$ eventually $\hat{S} \neq null \wedge \hat{z} \neq \infty$

A fixed point for this program is reached when all feasible covers have been generated. At this point, the program repeatedly assigns $seq = null$ and $v = n$.

**fixed point**

$$\text{FP} \equiv (seq = null \wedge v = n)$$

**Progress Condition:** The invariant ensures that a $seq$ is never repeated and always increasing until all feasible covers have been generated. The optimal cover is updated whenever a better cover is found. The last assignment statement in the program ensures that the fixed point is reached and maintained whenever all feasible covers have been explored. Thus, this UNITY program continually finds covering sets and accumulates the best cover and cost until the fixed point is reached.

*3.3.2.3 Parallel SCP UNITY Program* The previous UNITY program describes a serial branch-and-bound SCP. The serial version is developed first because the parallel version of the SCP is based on the serial version. The serial UNITY program is, therefore, mapped to a parallel UNITY program. In the following program, let $k$ be the index to $N$ processors:

**Program** *Parallel Branch-and-Bound SCP*

**declare**

| | | |
|---|---|---|
| $C[v]$ | : int | {cost of covering set $v$} |
| $k$ | : int | {processor index} |
| $l_k$ | : int | {calculated lower bound} |
| $N$ | : int | {number of processors} |
| $v_k$ | : int | {index variable} |
| $z_k$ | : int | {cost of the current state of the search on a processor} |
| $\hat{z}$ | : int | {global best cost} |
| $cover_k$ | : boolean | {TRUE if a cover exists} |
| $D$ | : boolean | {TRUE if a dominate set exists} |
| $L_k$ | : boolean | {TRUE if lower bound is exceeded} |
| $Iseq_k$ | : set | {starting state for processor $k$} |
| $seq$ | : set | {global list of subsequences} |
| $seq_k$ | : set | {current state of the search} |
| $\hat{S}$ | : set | {global best cover} |

**always**

$1 \leq k \leq N \wedge 1 \leq v_k \leq n$

$z_k = \langle +i : i \in seq_k :: C[i] \rangle$

$\hat{z} = z_k$ if $z_k < \hat{z} \wedge cover_k \sim$

$\hat{z}$

$seq = \langle \| \; k : 1 \leq k \leq N :: seq_k \rangle$

$\langle k : 1 \leq k \leq N :: Iseq_k \neq Iseq_{k+1} \rangle$

$\langle k : 1 \leq k \leq N :: seq_k = Iseq_k \cup seq_k \rangle$

$\hat{S} = seq_k$ if $z_k < \hat{z} \wedge cover_k$

$D(v_k) =$ TRUE if $seq_k \cup v_k$ is dominated by a previous $seq \sim$

FALSE

$L_k(v_k) =$ TRUE if $z_k + C[v_k] \geq l_k \sim$

FALSE

**initially**

$\hat{z} = \infty \; \| \; \hat{S}, seq = null, null$

$\| \; \langle \| \; k : 1 \leq k \leq N :: l_k, v_k, z_k, seq_k = 0, top(seq) + 1, C[seq], Iseq_k \rangle$

**assign**

| | | | |
|---|---|---|---|
| $seq_k,$ | $v_k$ | := | |
| $seq_k; v_k,$ | $v_k + 1$ | if | $z_k + C[v_k] < \hat{z} \wedge \overline{cover_k} \wedge \overline{D(seq_k; v_k)} \wedge \overline{L_k(seq_k; v_k)} \sim$ |
| pop($seq_k$), top($seq_k$)+1 | | if | $seq_k \neq null \wedge (z_k + C[v_k] \geq \hat{z} \vee cover_k \vee D(seq_k; v_k) \vee L_k(seq_k; v_k)) \sim$ |
| $null,$ | $n$ | if | $seq_k = null \wedge (z_k + C[v_k] \geq \hat{z} \vee cover_k \vee D(seq_k; v_k) \vee L_k(seq_k; v_k))$ |

**End** {*Parallel Branch-and-Bound SCP*}

The mapping of the serial program to a parallel program is accomplished using data parallelism discussed in Section 3.3.1. The search graph is partitioned between all processors such that each processor searches a distinct section of the graph shown in Figure 3.7.

There are three major differences between this program's always-section and the previous program's. First, many of the variables are now specific to a processor with the addition of a subscript $k$. Furthermore, two statements have been added to ensure that a processor's initial branch of the search graph ($Iseg_k$) is distinct and that the processor's initial branch is always included in its list of covering sets ($seg_k$). The final addition keeps the global best cost ($\hat{z}$) and cover ($\hat{S}$) up to date.

Since global data structures are employed in the parallel program and the input search graph is partitioned to the processors, additional statements are added to the initially-section. Statements are added to initialize each processor's list of covering sets, its cost, and its indices.

The parallel program's assign-section operates the same as the serial program's assign-section. The only difference is that two of the bounding functions are now global. The cost of the covering sets, $z_k$, is compared to a global best cost, $\hat{z}$, rather than a local best cost and the dominance test compares a processor's sequence against a global list of sequences. Hence, the best cost and unique subsequences obtained by any processor are used by all processors to bound their search graph.

One of UNITY's strengths is its support of an iterative development design technique and its ability to rely on previous assertions in the proof system. The invariant remains the same but the fixed point is modified to control the additional $N$ processors. A fixed point for the parallel program is reached when all feasible covers have been generated by all processors. At this point, each processor's program repeatedly assigns $seq_k = null$ and $v_k = n$.

**fixed point**

$$\text{FP} \equiv \langle \| \, k : 1 \leq k \leq N :: seq_k = null \wedge v_k = n \rangle$$

The design now includes high-level algorithms to construct the table and partition the input search tree so that all searching processors receive a subbranch of the tree to search. Furthermore, serial and parallel UNITY programs exist to search the subbranches. The UNITY programs are based on a branch-and-bound search of the state space and include four bounding functions. Pseudo-code algorithms based on the UNITY programs are presented in the next chapter. The next section presents high-level parallel algorithms for the general reductions discussed in Section E.6.4.

*3.3.3 Reduction Techniques* From Section E.6.4, recall that reduction #1 simply checks whether a solution exists. Given the vertex record, the algorithm looks at the bottom of the list of vertices and, if the number of 1's in the last row is zero, a solution does not exist and no search is initiated. Reduction #2 removes all vertices that are covered by only one set, adds the corresponding sets to the list of covering sets, and removes the covering sets. This algorithm starts at the bottom of the list of vertices and moves up the list continually examining the vertex cardinality field for rows containing only one covering set. If such a row exists, the algorithm finds the covering set by indexing into the matrix. The rows covered by this set are then removed from the matrix along with the set. At this point, the indices into the matrix are incorrect; therefore, it becomes necessary to add an additional field to the vertex and set records. The new field (`True Vertex` and `True Set`) contains the row/column of the original, unreduced matrix. The original **Index** field of the records is now used to index into the reduced matrix. Hence, this field is renumbered if any rows or columns are removed. Figures 3.8 and 3.9 show the "newly" created vertex and set vectors. Reductions #1 and #2 are relatively simple given the specified vertex and set vectors; hence, these reductions are not parallelized. They are conducted in the controlling processor a priori.

Reductions #3 and #4 are more complicated and could potentially benefit from a parallel implementation. The parallelization of these reductions is accomplished by evenly dividing the vertex or set record between the processors and then marking the rows or columns to be removed. All processors must receive the same reduced matrix; thus, any rows and columns to be removed are removed in the controlling processor which then

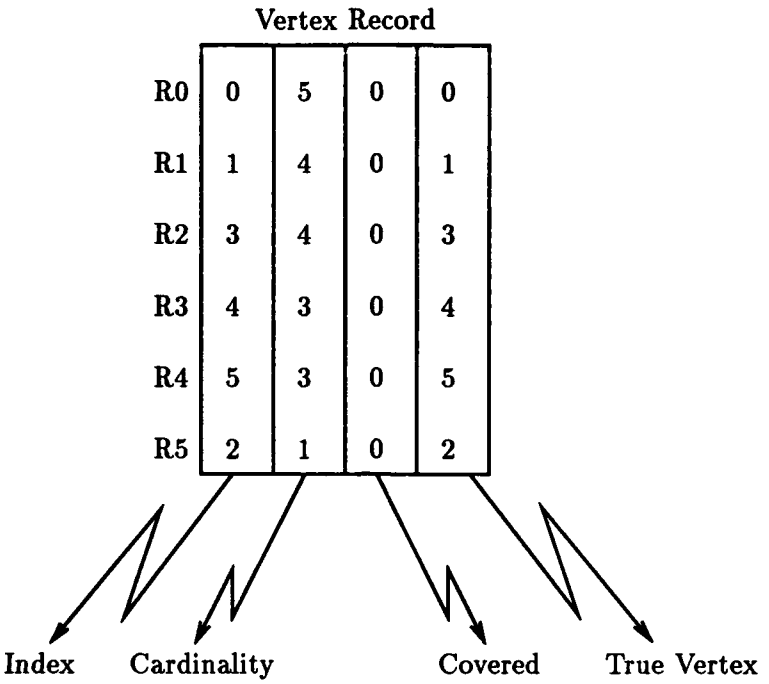



Figure 3.8. New Vertex Record

Set Record

| C0 | C1 | C2 | C3 | C4 | C5 | C6 | C7 | |
|---|---|---|---|---|---|---|---|---|
| 5 | 4 | 0 | 2 | 7 | 6 | 1 | 3 | Index |
| 2 | 3 | 4 | 5 | 5 | 6 | 7 | 8 | Cost |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | Used |
| 5 | 4 | 0 | 2 | 7 | 6 | 1 | 3 | True Set |

Figure 3.9. New Set Record

broadcasts the reduced matrix to the searching processors.

Reduction #3 checks for dominated vertices and reduction #4 checks for dominated sets as described in Section E.6.4 on page E-10. The algorithm is the same for both reductions; therefore, only reduction #3 is explained. The check for dominated vertices requires a comparison between all rows of the matrix. The algorithm selects the first row and compares it to the following rows as indexed by the vertex vector. If any of the following rows dominates the first row, the first row is marked in the covered field of the vertex record and the algorithm immediately selects the second row for comparison. If, however, the first row dominates any of the following rows, these rows are marked for removal and the algorithm continues searching the rows for other dominated rows. Once all dominated rows are found, the rows marked as covered are removed from the 0–1 matrix and vertex vector. An algorithm is presented in the next chapter.

Reductions #3 and #4 are easily accomplished in a parallel architecture. The entire 0–1 matrix of say $N \times M$ elements is distributed to all processors and the vertex vector is divided evenly among $2^n$ processors for an even number of rows (i.e., $N$ is divisible by 2). In the case of an odd number of rows, the lower address processors (e.g., processor 0, processor 1) receive one extra vertex record. The reduction algorithm for each processor checks and marks its initial subset of the vertex vector. Following this, the processors combine the subsets in a logarithmic fashion[1] until the marked vertex vector is reassembled.

For example, let there be four processors numbered 0, 1, 2, and 3 in a parallel architecture such as a hypercube and let the vertex vector consist of eight vertex records numbered 0 through 7. The reduction algorithm assigns records 0 and 1 to processor 0, records 2 and 3 to processor 1, and so forth. Each processor marks the dominated vertices in its subset of the vertex vector. Processor 0 now receives the vertex vector from processor 1 and processor 2 receives the vertex vector from processor 3. The vector is once again checked for dominated vertices skipping those vertices already marked. Finally, processor 0 receives the vertex vector from processor 2 and checks for dominated vertices skipping those

[1] Combining sets in a logarithmic fashion means to combine the sets of nearest neighbor processors. The sets are efficiently recombined in this manner since communication is minimal between nearest neighbors in a hypercube.

vertices already marked for removal. The number of iterations is thus $\log_2(2^n) = n$ for a cube of dimension $n$. An algorithm is presented in the next chapter.

### *3.4 Complexity Analysis*

Now that the initial control and data structures have been developed, it is possible to conduct a complexity analysis. This analysis is divided into a worst and a best case. The worst case assumes a serial processor whereas the best case is derived from a mythical architecture specially suited to the SCP. The mythical architecture is described in the best case analysis.

*3.4.1 Worst Case* Consider a serial computer searching for the optimal set cover. It chooses a subset of the total set of covering sets until all possible subsets are checked. For example, let $S = \{1, 2, 3\}$ represent the set of covering sets. Now, generate all possible subsets of $S$: $T = \{\phi, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}\}$. Such an enumeration of all possible sets is called a power set ($T = \mathcal{P}(S)$) and contains $2^n$ elements (54:104). In the worst case, the search algorithm must check all possible subsets of the complete set of covering sets (excluding the empty set). If there are *nsets* number of covering sets, the worst case search generates a power set of the covering sets and is $O(2^{nsets})$.

*3.4.2 Best Case* Consider a hypothetical parallel architecture with $2^{nsets}$ processors. Furthermore, let the interconnection network allow simultaneous transmission of all data from a central controller to all processors and the intraprocessor connection network be structured as a tree. The algorithm on each of the $2^{nsets}$ processors checks a specific subset of the $2^{nsets}$ subsets such that no two processors check the same subset. One of the processors must combine all covering sets to compute the cost for subset $T = \{1, 2, 3, \ldots, nsets\}$. The worst case order-of for $T$ is $O(nsets)$ since one processor must include all *nsets* elements in its cover.

Since the central controller can not compare all covering sets simultaneously to obtain the optimal cover, the $2^{nsets}$ processors must do the comparison until the optimal cover is found. Given the specified interconnection network, the comparison is structured

as a binary tree. To find the optimal cover now requires $O(\log_2(2^{nsets})$ or $O(nsets)$ comparisons (i.e., communications). The best case solution to the SCP using the hypothetical architecture is then $O(nsets^2)$.

*3.4.3 iPSC/2* Consider a cube-connected computer such as the iPSC/2 hypercube. The iPSC/2's communications network allows a broadcast to all processor nodes via an Ethernet TCP/IP local area network (37:1-12). This network approximates the above best case specification to transmit to all nodes simultaneously. Furthermore, the cube network easily models a binary tree network; hence, the node processors are capable of obtaining the $O(\log_2(2^{nsets})$ or $O(nsets)$ communications. Even so, the iPSC/2 consists of, at most, $2^7 = 128$ processors. Assuming an algorithm is designed as in the above best case complexity analysis, the largest problem the iPSC/2 can solve is an $N \times 7$ matrix. Note that even though it is possible to theoretically obtain this order-of on a realistic computer, the implementation is clearly not an efficient use of the hardware.

## *3.5 Computational Equipment*

High-level serial and parallel SCP algorithms have been designed and a complexity analysis performed on the SCP. One of the goals of this research is to implement a parallel SCP algorithm on a parallel architecture. Clearly, the mythical architecture specified in the previous analysis is the ideal parallel computer for this implementation. The mythical computer does not exist; however, AFIT currently houses three hypercube computers which are distributed memory computers similar is some respects to the mythical computer. For instance, the network of a hypercube computer is suitable for implementing a tree search as discussed in Section 2.5.2.

Two of AFIT's hypercubes are Intel iPSC/1 models with 32 nodes each and the other hypercube is an iPSC/2 model with 8 nodes. All three are available for this research; however, the iPSC/2 is a later and more advanced hypercube containing faster hardware and more software functions. Hence, the programs for this research are implemented on the iPSC/2. The following two sections briefly describe the iPSC/2's hardware and software.

*3.5.1 Intel iPSC/2 Hypercube* The Intel iPSC/2 is a cube-connected supercomputer containing between eight and 128 nodes. Each of the nodes contains an 80386 microcomputer, an 80387 numeric coprocessor, an optional SX scalar processor module based on the Weitek 1167 floating-point unit, an optional VX vector processor, and 1–16 megabytes of memory. The VX vector processor is an extension to the cube and allows the nodes to achieve more than 400[2] million floating-point operations per second (Mflops). The host contains an 80386 and 80387, eight megabytes of memory, and a 140 megabyte harddisk (37). AFIT's current configuration is summarized in Table 3.1. The node processors communicate with each other over Intel's proprietary communications network called a Direct-Connect™ network. The network supports simultaneous node-to-node communication with uniform communication speed between all nodes. Although the nodes are physically connected as a hypercube, any communication between nodes is controlled in the hardware. As such, the message path between two communicating nodes is established and the message is passed by dedicated interface modules. The node's main processor does not

Table 3.1. AFIT's iPSC/2 Hypercube Configuration (36, 37, 33)

| | |
|---|---|
| CPU | Intel 80386, 32-bit |
| Math Coprocessor | Intel 80387 |
| Numeric Accelerators | SX Scalar Processor |
| Clock | 16 MHz |
| Operating System | Host: AT&T UNIX, Version V |
| | Node: NX/2 |
| Hard Disk | 140 Mbytes |
| Memory | Host: 8 Mbytes |
| | Node: 4 Mbytes |
| Number of Nodes | 8 |
| Cube Network | Direct-Connect™ Routing<br>2.8 MBytes/sec. bandwidth |
| Max Message Length | Host↔Node: 256 KBytes |
| | Node↔Node: Memory Size |
| Max Processes Per Node | 20 |

[2]Compare to the iPSC/860 recently released by Intel. The iPSC/860 is an expandable supercomputer based on RISC technology. The system consists of 8 to 128 processors and performance ranges from 480 Mflops to 7.6 Gigaflops.

cease processing to assist in the communication as is the case with the iPSC/1 hypercube. Once the link between the communicating nodes is established, the message is transmitted at 2.8 megabytes per second. This dedicated interface hardware is fast enough to give the appearance of a direct connection between all nodes. The host processor contains the development environment which consists of AT&T UNIX System V, FORTRAN, C, and LISP compilers (32).

*3.5.2 Software* AFIT's version of the iPSC/2 contains C, LISP, and FORTRAN compilers. Either language may be used to implement the SCP algorithms; however, since the C language contains efficient constructs for manipulating pointers, the SCP algorithms are written in C. The Green Hills C Compiler hosted on the iPSC/2 is an optimizing compiler which generates full 32-bit object code and supports standard UNIX functions as well as cube functions supplied in libraries (34). Many of the programs written for the SCP are serial; hence, much of the initial code development is conducted on a personal computer using Borland's Turbo C 2.0 which supports interactive editing, compiling, and debugging. In addition to the serial programs, many of the parallel programs are written and compiled under Turbo C. These parallel programs, however, are not tested with Turbo C because the software required to simulate a hypercube computer is tedious and difficult to develop. However, the use of Turbo C facilitates quick correction of any compile-time errors. Standard K&R C (39) is used throughout the coding. Green Hills C and Turbo C are sufficiently compatible that many of the programs written with Turbo C may be directly ported to the Green Hills C compiler. As such, few problems were found during the rehosting of this software.

The software engineering practice is discussed in Section 1.5. All the programs contain common headers shown in Figures 3.10 and 3.11. The file header, Figure 3.10, is located at the top of each file; whereas, each routine in the file contains the header shown in Figure 3.11. Furthermore, structure charts and abstract data types are contained in Appendices A and B.

```
/*****************************************************************************
*  DATE:  mm/dd/yy of last change to this file
*  VERSION:  x.y where x represents new modules added or deleted and
*                    y represents changes within a module
*
*  TITLE:  Title given to this file.  English title, not filename.
*  FILENAME:  Intended filename and extension.
*  COORDINATOR:  Who is responsible for this file.
*  PROJECT:  Name of the software project of which this file is a part.
*  COMPUTER:  If a special computer is required.
*  OPERATING SYSTEM:  Name and version number of OS under which this file
*                     was developed to run.
*  LANGUAGE:  Name and version of compiler or assembler under which this
*             file was written.
*  FILE PROCESSING:  How this file is used.   Is it INCLUDED in another
*                    file?  Is it compiled and/or assembled?  What files
*                    must it be linked with?  What compiler options must
*                    be specified?
*  CONTENTS:  What modules are contained in this file?  Specify number and
*             name.  Include one line functional description.
*  FUNCTION:  Briefly, what is the overall function of this entire file?
*
*****************************************************************************/
```

Figure 3.10. Software Documentation File Header

```
/*****************************************************************************
*
*  DATE:  mm/dd/yy -- date of this version
*  VERSION:  x.y where x represents new modules added or deleted and
*                      y represents changes within a module
*
*  NAME:  Program name -- English name
*  MODULE NUMBER:  Module number in current program.
*  DESCRIPTION:  Text description of the module's function.
*  ALGORITHM:  Algorithm used.
*  PASSED VARIABLES:  Variable name(in), (out), (in/out)
*  RETURNS:  Value returned by this module.
*  PROTOTYPE:  ANSI prototype for this routine.
*  GLOBAL VARIABLES USED:  Those read by this module.
*  GLOBAL VARIABLES CHANGED:  Those changed by this module.
*  FILES READ:  Files read by this module.
*  FILES WRITTEN:  Files written or appended by this module.
*  HARDWARE INPUT:  I/O ports or devices read.
*  HARDWARE OUTPUT:  I/O ports or devices written.
*  MODULES CALLED:  Other procedures or subroutines called.
*  INCLUDED FILES:  #include files
*                   local.h -- defines some commonly used variables
*
*  AUTHOR:  Person who wrote this version.
*  HISTORY:  Name, author, and date of earlier version.
*            What changes were made?
*
*  ANALYSIS:  Time and space complexity.
*
*****************************************************************************/
```

Figure 3.11. Software Documentation Module Header

## *3.6 Summary*

This chapter has presented the detailed plan of attack, conducted the preliminary design for the search algorithms as well as the preconditioning and reduction techniques, analyzed the best and worst case execution, and finally described the hardware and software used for the implementation. The preliminary design of the search algorithm identified a method to employ for the parallel decomposition of the SCP. The data is partitioned using a coarse grain algorithm with static allocation of the initial search tree and communication with a central processor to transmit and receive the global best cost and covering sets. Parallel algorithms for the reduction techniques described in Section E.6.4 are also presented. The next chapter contains a detailed design of the parallel SCP programs including the design of the dominance and lower bound tests.

# *IV. Detailed SCP Program Design & Implementation*

## *4.1 Introduction*

The previous chapter discusses the research methodology and develops a preliminary design for the construction of the SCP table and allocation of subbranches to searching processors. The preliminary design of the SCP parallel programs contains two UNITY metaprograms[1]. The first metaprogram is a serial branch-and-bound SCP and the second metaprogram is a parallel branch-and-bound SCP. Many details remain before the design is implemented; hence, this chapter contains a detailed design of the the algorithms for a serial version and three parallel versions of the SCP.

Section 4.2 maps the UNITY metaprograms developed in Chapter III to the iPSC/2 hypercube and explains the algorithms and data structures which perform the search. The remaining sections, Sections 4.3, 4.4, 4.5, and 4.6 explain the algorithms and data structures for the dominance test, the lower bound test, the reduction techniques, and the parallel bitonic merge sort.

## *4.2 Mapping of the Search Methodology*

This section develops one last UNITY metaprogram and presents pseudo-code algorithms for the serial and parallel versions of the SCP.

*4.2.1 UNITY Mapping* The UNITY metaprograms presented in Sections 3.3.2.2 and 3.3.2.3 characterize the SCP as a serial and a parallel search respectively. According to Chandy and Misra's design methodology, the next step in the design process is to map the UNITY program to a target architecture (iPSC/2 for this research) using program schemas.

A mapping is "an architecture specific implementation of a UNITY program" mapped to the following three architectures (16:82):

[1] As discussed on page 2-21, UNITY programs are not programs in the classical sense of the word. A UNITY program is better thought of as a design representation of a program. The UNITY designs are referred to as programs since this is the convention established by Chandy and Misra (16)

**Asynchronous shared-memory architecture** — "Consists of a fixed set of processors and a fixed set of memories" (16:82).

**Distributed systems** — "Consists of a fixed set of processors, a fixed set of channels, and a local memory for each processor" (16:83).

**Synchronous architectures** — Same as the asynchronous shared-memory architecture with each processor sharing a common clock (16:84).

A program schema is "a restricted class of UNITY programs and associated mappings" defined as follows (16:82):

**Read-only schema** — "A UNITY program and its mapping to an asynchronous shared-memory computer is in the read-only schema if each variable in the program is modified by (statements in) at most one processor. Programs in the read-only schema can be executed on architectures in which each memory is written into by at most one processor" (16:85).

**Shared-variable schema** — "A program and mapping fit the shared-variable schema if each statement (allocated to a processor) names at most one nonlocal variable. The nonlocal variable may appear on the left or the right side of the statement. This implementation employs locks on shared variables. At most one processor holds a lock on a shared variable at any time" (16:86).

**Equational schema** — "A program in the equational schema is a *proper* set of equations, consisting of only the declare-section and the always-sections" (16:87).

**Single-statement schema** — "A program is in the single-statement schema if the assign-section of the program consists of a single statement (which may be a multiple assignment)" (16:88).

Since this program is implemented on a cube-connected architecture, the mapping is to a distributed system using a shared-variable schema. The parallel UNITY program of Chapter III, *Parallel Branch-and-Bound SCP*, requires one modification to map it to the iPSC/2. Recall that the previous UNITY program implements two global variables;

namely, a global best cost ($\hat{z}$) and the list of dominating sets ($L$). Since the iPSC/2 is a distributed architecture, the global data structures are maintained and updated on a single processor. Therefore, any updates to or information obtained from the global data structures is accomplished over the iPSC/2 interconnection network. The global best cost consists of a single integer; therefore, update and broadcast of $\hat{z}$ is not likely to create a communications bottleneck. On the other hand, the list of dominating sets is updated many times throughout the entire execution of the program and the list does not consist of one integer as the global cost does, but may contain as many as $n-1$ integers. The constant broadcast of this information would quickly swamp the network. Therefore, to minimize the communications overhead, each processor maintains its own list of dominating sets based on its search and transmits/receives a globally maintained best cost.

The UNITY program on the next page maps the parallel branch-and-bound SCP to a distributed, cube-connected architecture. The major modifications consist of removing the global list of subsequences, *seq*, and assigning $D$ to specific processors.

Specification of this mapping completes the UNITY design process. To recap, a serial program is developed first as in Chapter III and is informally proven correct using an invariant, a fixed point, and a progress condition. The serial program is then transformed to a parallel UNITY program by dividing the input data structure among the processors. An informal proof of the parallel version follows from the proof of the serial version. And finally, the parallel program is mapped to a distributed computer in the previous paragraphs.

The previous chapter presents the preliminary design a parallel SCP algorithm based on data parallelism. Typically, data parallelism divides the data between the processors which execute essentially serial algorithms; hence, a parallel version of the SCP includes many serial subroutines. To aid in the parallel implementation and to assist in testing, a serial version of the SCP is designed and implemented first. Then, three successively more complex versions of the parallel SCP programs are developed with each new version utilizing the work accomplished in the previous version. The pseudo-code algorithms in this chapter are supplemented by structure charts and ADTs contained in Appendices A and B in accordance with the software engineering practices outlined in Section 1.5.

**Program** *Distributed System Parallel Branch-and-Bound SCP*

**declare**

| | | |
|---|---|---|
| $C[v]$ | : int | {cost of covering set $v$} |
| $k$ | : int | {processor index} |
| $l_k$ | : int | {calculated lower bound} |
| $N$ | : int | {number of processors} |
| $v_k$ | : int | {general index variable} |
| $z_k$ | : int | {cost of the current state of the search on a processor} |
| $\hat{z}$ | : int | {global best cost} |
| $cover_k$ | : boolean | {TRUE if a cover exists} |
| $D_k$ | : boolean | {TRUE if a dominate set exists} |
| $L_k$ | : boolean | {TRUE if lower bound is exceeded} |
| $Iseq_k$ | : set | {starting state for processor $k$} |
| $seq_k$ | : set | {current state of the search} |
| $\hat{S}$ | : set | {global best cover} |

**always**

$1 \leq k \leq N \wedge 1 \leq v_k \leq n$

$z_k = \langle +i : i \in seq_k :: C[i] \rangle$ {Cost of the cover on processor $k$}

$\hat{z} = z_k$ if $z_k < \hat{z} \wedge cover_k \sim$ {Maintains a current global best cost}
$\hat{z}$

$\langle k : 1 \leq k \leq N :: Iseq_k \neq Iseq_{k+1} \rangle$

$\langle k : 1 \leq k \leq N :: seq_k = Iseq_k \cup seq_k \rangle$

$\hat{S} = seq_k$ if $z_k < \hat{z} \wedge cover_k \sim$ {Maintains the current best cover}
$\hat{S}$

{Dominace test on processor $k$}

$D_k(v_k) =$ TRUE if $seq_k \cup v_k$ is dominated by a previous $seq_k \sim$
FALSE

{Lower bound test on processor $k$}

$L_k(v_k) =$ TRUE if $z_k + C[v_k] \geq l_k \sim$
FALSE

**initially** {Initialize all variables}

$\hat{z} = \infty \parallel \hat{S} = null$
$\parallel \langle \parallel k : 1 \leq k \leq N :: l_k, v_k, z_k, seq_k = 0, top(seq) + 1, C[seq], Iseq_k \rangle$

**assign**

| | | | |
|---|---|---|---|
| $seq_k,$ | $v_k$ | := | |
| {Branch forward} | | | |
| $seq_k; v_k,$ | $v_k + 1$ | if | $z_k + C[v_k] < \hat{z} \wedge \overline{cover_k} \wedge \overline{D_k(seq_k; v_k)} \wedge \overline{L_k(seq_k; v_k)} \sim$ |
| {Backtrack} | | | |
| pop($seq_k$), top($seq_k$)+1 | | if | $seq_k \neq null \wedge (z_k + C[v_k] \geq \hat{z} \vee cover_k \vee D_k(seq_k; v_k) \vee L_k(seq_k; v_k)) \sim$ |
| {Final state} | | | |
| $null,$ | $n$ | if | $seq_k = null \wedge (z_k + C[v_k] \geq \hat{z} \vee cover_k \vee D_k(seq_k; v_k) \vee L_k(seq_k; v_k))$ |

**End** {*Distributed System Parallel Branch-and-Bound SCP*}

*4.2.2 Serial SCP* This section develops the pseudo-code algorithms for a serial implementation of the SCP. A high-level algorithm for the overall SCP search process is presented first followed by an explanation of the various data structures and algorithms selected to complete the entire search process. The following algorithm controls the general search process (17:40–43):

**Algorithm** *SCP*
    Read the 0-1 matrix
    Reduce 0–1 matrix
        Reduction #1
        Reduction #2
        Reduction #3
        Reduction #4
    Construct the table
    Search the table for an optimal solution
        Dominance Test
        Lower Bound Test
    Display results
**End** *SCP*

The structure charts for the serial SCP programs are contained in Section A.1 and the abstract data types (ADTs) are given in Section A.2. The structure chart and ADT for this particular algorithm are given in Figure A.1 and ADT A.10.

The first line in the SCP algorithm is a simple file input routine that reads the 0–1 matrix. The input file must contain the number of rows and columns in the matrix, the matrix, and the costs of the columns. An example input matrix is shown in Figure 4.1. The first seven rows comprise the file header and may contain any number of rows preceded by #. The number of matrix rows (6) and columns (8) is read into the program variables **Nverts** and **Nsets** respectively. These values are used to establish the base data structures: **AdjMat**, **Vertex**, and **Set**. As the file is read, the base data structures are constructed. **AdjMat** is a two-dimensional 0–1 matrix. **Vertex** and **Set** are the vertex and set vectors shown in Figures 3.8 and 3.9 on page 3-25.

Each of these base data structures is dynamically allocated; hence, the programs only allocate the amount of memory necessary to contain the data. Dynamic memory allocation increases the flexibility of the programs by not constraining the size of the input

```
#
#           Filename: dummy.dat
#           Number of Vertices: 6
#           Number of Sets: 8
#           Density of 1's: 0.416667
#           Cost range: [2,8]
#
6
8
11000010
10100101
00001110
00010000
01001011
11100101
4 7 5 8 3 2 6 5
```

Figure 4.1. Input File Format for the SCP

problem and it decreases the amount of unused memory. Memory usage is a concern because the node processors on the iPSC/2 contain limited memory, reference Table 3.1, and no virtual memory. Since the C language is so intimately linked with pointers to data structures (39), dynamic data structures referenced by pointers should not slow the execution of the program. However, the control syntax is complicated. Figures 4.2 and 4.3 show the **Vertex** and **Set** data structures in C syntax.

```
#define VERTEX struct vertex_type
VERTEX {
    int Index,          /* The index into AdjMat. */
        Card,           /* The cardinality of the true vertex. */
        Covered,        /* Signals this vertex as covered. */
        TrueVertex;     /* The true vertex of the original AdjMat. */
};
```

Figure 4.2. Language Defined Vertex Data Structure

Once the base data structures have been constructed, the reductions (Section 3.3.3) are performed if requested by the user. Since the reductions are not required for the search process, their pseudo-code algorithms are presented later in Section 4.5.

```
#define SET struct set_type
SET {
    int Index,      /* The index into AdjMat. */
        Cost,       /* The cost of the true set. */
        Used,       /* This set has been used already. */
        TrueSet;    /* The true set of the original AdjMat. */
};
```

Figure 4.3. Language Defined Set Data Structure

```
#define TABLE_NODE struct table_node
TABLE_NODE {                    /* The structure of one node of the table. */
    int Row,                    /* The row of Vertex. */
        Col;                    /* The set of Set. */
    TABLE_NODE      *Down,      /* Pointer to the next block in the table. */
                    *Right;     /* Pointer to the right node in the table. */
};
```

Figure 4.4. Language Defined Table Data Structure

Following the reductions, the table must be constructed. The data structure for the table was illustrated in Figure 3.3 on page 3-11 and the C syntax is given in Figure 4.4. As with the base data structures, the table is also dynamically allocated and occupies only the memory required. The following algorithm describes the construction of the table:

**Algorithm** *Build Table*
    Point to the first row in the vertex vector
    Build a block for each row in the vertex vector
        Loop on the cardinality of this row
            Construct a **TABLE_NODE** for each column covering this row
        End loop
        Point to the next row
    End for
**End** *Build Table*

Since the **Vertex** and **Set** vectors are sorted in ascending order and index into the 0–1 matrix, the construction of the table data structure follows a simple algorithm of building a block for each row of the **Vertex** vector.

The following algorithm searches the table for an optimal cover and a structure chart is given in Figure A.5:

```
Algorithm Serial SCP Search
    Present cost = cost of next set = 0, best cost = ∞
    Loop until done
        Loop while all rows not covered and present cost + cost of next column < best cost
            Look for the next uncovered row
            Look for the next unused column in this block
            If current state is dominated by a previous state
                Exit loop (Backtrack)
            End if
            If lower bound of current state exceeds current cost
                Exit loop (Backtrack)
            End if
            Save column on the stack
            Update present cost
            Mark the column and all rows covered by the column
        End loop
        If all rows are covered and present cost < best cost
            Save the columns and cost (SOLUTION)
        End if
        Loop (BACKTRACK)
            Pop a column off the stack
            Remove the column from the solution and present cost
            Mark the rows as UNCOVERED if not covered by another column
        End loop
        If the stack is empty, then the search is complete
            Return the best cover
        End if
    End loop
End Serial SCP Search
```

Notice from the algorithm that a stack is used to store the current state of the search. The structure chart for the stack is shown in Figure A.6 while a pictorial and syntactical representation of the data structure is shown in Figures 4.5 and 4.6, respectively. This stack is a semi-generic stack constructed as a singly linked-list that points into the table data structure.

Aside from the reductions, the dominance test, and the lower bound test described later in this chapter, the detailed design of the serial SCP algorithm is complete. The next section describes the algorithms and data structures for the parallel implementations of the SCP.

*4.2.3 Parallel SCP* Three increasingly complex parallel SCP algorithms are developed in the following sections. Corresponding structure charts and ADTs are presented in Sections B.1 and B.2. The three versions of the SCP differ mainly in their load balancing

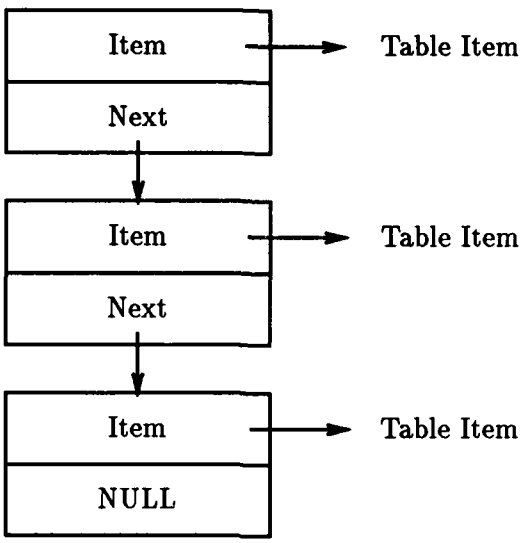



Figure 4.5. Stack Data Structure

```
#define S_NODE struct s_node
S_NODE {                    /* The structure of a node in the stack. */
    S_NODE *Next;           /* Pointer to the next node in the stack. */
    TABLE_NODE *Item;       /* Pointer to a node in the table. */
};
```

Figure 4.6. Language Defined Stack Data Structure

schemes. The first algorithm uses a static allocation of the initial search graph similar to the algorithms designed in Section 3.3.1. The initial graph is obtained by doing a breadth-first expansion, by the searching processors, on the nodes of the graph until $m$ subgraphs have been developed where $m$ is defined as the number of searching processors. The second algorithm improves on the static allocation scheme by moving construction of the subgraphs to the controlling processor and dynamically assigning subgraphs to the searching processors as they finish their current search. The final algorithm adds a dynamic load balancing scheme to the previous algorithm. In this version, the processors dynamically share portions of their respective search graphs until every searcher terminates.

The basic design of all three parallel algorithms designates one processor (the controller) to control all other searching processors (the searchers). The interaction between the two processor designations is specified by the following algorithms:

```
Algorithm Generic SCP Controller
    Loop until all searchers finished
        Poll receive buffer for new global best cost or searcher finished
        If new global best cost
            Extract the cost and cover from the received list
            Compare the cost to the previous global best cost
            Retain the cover and cost with the lowest cost
            Broadcast new global best cost to all searchers
        End if
    End loop
    Poll searchers for performance data
    Send results to host processor
End Generic SCP Controller

Algorithm Generic SCP Searcher
    Loop until search complete
        Poll receive buffer for a new global best cost
        Search for a cover
        If a cover found and its cost is less than global best cost
            Transmit global best cost and cover to controller
        End if
        Continue search
    End loop
    Send performance data to controller
End Generic SCP Searcher
```

The controller continually polls its receive buffer for the presence of a new global best cost and associated cover or for the termination of a searcher. In the course of its

search, a searcher submits a global best cost and cover to the controller if it finds a cover with a lower cost. The controller compares the new global best cost to the currently held global best cost and retains the minimum cost and associated cover. It then broadcasts the new global best cost to all searchers. The searchers, on the other hand, periodically poll their receive buffers for a new global best cost. If one is received and it has a lower cost, the processor immediately updates its locally maintained copy of the global best cost. When all searchers terminate, the controller polls each searcher for its performance data and then sends the optimal cover, the optimal cost, and the performance data to the host for display to the user.

On the iPSC/2 computer, it is difficult to provide a suitable user interface directly with the node processors; hence, the node programs usually interface with a program executing on the host processor (37:1-1). The host program contains the user interface, loads the node processors with the proper version of the SCP algorithm and displays the results. The following algorithm executes on the host processor and is common to all parallel versions of the SCP:

**Algorithm** *SCP Host*
    Read the matrix
    Construct the base data structures: **AdjMat**, **Vertex**, and **Set**
    Broadcast **AdjMat**, **Vertex**, and **Set** to the controller
    Wait for the optimal cover and performance data to return
    Display the results
**End** *SCP Host*

The host reads the input 0–1 matrix and builds **AdjMat**, **Vertex**, and **Set**. These data structures are then transmitted to the controller for disposition. The structure chart in Figure B.1 illustrates the communication channels between this host algorithm, the controller algorithms, the searcher algorithms, and the dynamic load balance algorithm.

The following sections detail the three parallel algorithms starting with the statically allocated coarse grain algorithm, followed by the fine grain algorithm, and finally, the dynamic load balanced parallel version.

*4.2.3.1 Statically Allocated Coarse Grain Algorithms* A statically allocated coarse grain algorithm is developed first since it is the simplest to design and implement

with much of the design developed and explained in the previous chapter. The structure charts for the controller and searcher algorithms are contained in Figures B.3 and B.4 and the ADTs are represented in ADTs B.18 and B.21.

**Algorithm** *Statically Allocated Parallel SCP Controller*
    Receive **AdjMat**, **Vertex**, and **Set** from the host
    Coordinate user requested parallel reductions
    Sort the **Vertex** and **Set** records
    Broadcast **AdjMat**, **Vertex**, and **Set** to the searchers
    Loop until all searchers finish
        If a better global best cost is received
            Save new global best cost and covering set
            Broadcast cost to all searchers
        End if
    End loop
    Poll searchers for performance data
    Send optimal covering set to host
    Send performance results to host
**End** *Statically Allocated Parallel SCP Controller*

The controller receives the base data structures from the host processor and then coordinates the requested parallel reductions, sorts the **Vertex** and **Set** vectors, and sends all data structures to the searchers. The parallel reductions are briefly explained in the previous chapter and detailed algorithms are presented in Section 4.5. The sort algorithm performs a quicksort for small input records and a parallel bitonic merge sort (reference Section 4.6 in this chapter) for large data sets. Once the initial processing of the data is finished, the controller enters a loop and waits for a searcher to finish or update the global best cost. Notice that the controller algorithm does not build the table nor does it partition any subbranches to the searchers. These tasks are performed by the searchers in this first parallel implementation.

Each searcher receives the matrix from the controller, constructs the table, and executes a breadth-first expansion until at least $m$ subgraphs exist. Note that each searcher performs the same steps so that all searchers contain copies of the matrix, the table, and the list of subgraphs.

An algorithm for the breadth-first expansion is given on page 3-13 in Section 3.3.1. The actual name of the expansion algorithm is *BuildStartingSets* as shown in ADT B.27. *NodeStartingSets* computes the number of subgraphs for each searcher and *NextStartingSet*

returns the next subgraph based on a searcher's processor ID (e.g., processor 0). All that remains is to specify the queue data structures used by the expansion algorithms. A structure chart and ADT for a semi-generic queue are given in Figure B.9 and ADT B.14. The queue data structure is illustrated in Figure 4.7 and the C syntax in Figure 4.8.

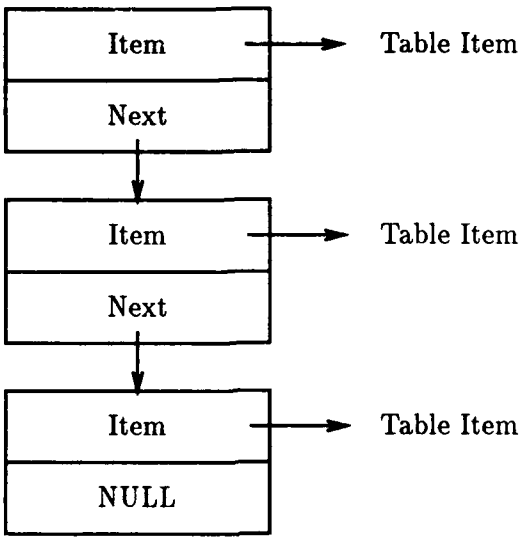



Figure 4.7. Queue Data Structure

```
#define Q_NODE struct q_node
Q_NODE {                    /* The structure of a node in the queue. */
    Q_NODE *Next;           /* Pointer to the next node in the queue. */
    TABLE_NODE *Item;       /* Pointer to a node in the table. */
};
```

Figure 4.8. Language Defined Queue Data Structure

When the expansion is complete, the searchers select predesignated subgraphs from their list of subgraphs. Consider three searching processors and eight subgraphs developed as a result of the breadth-first expansion on the search graph. The static allocation algorithm executed by every searcher assigns subgraphs {1, 4, 7} to processor 1, subgraphs {2, 5, 8} to processor 2, and subgraphs {3, 6} to processor 3.

The expanded subgraphs searched by a modified version of the *Serial SCP Search* algorithm described in Section 4.2.2. The parallel search utilizes a local and global best cost to increase the number of nodes pruned from the search graph; hence, the serial search algorithm is modified to include send and receive operations for the global best cost. The structure charts are given in Figures B.4 and B.8 and an ADT is given in ADT B.21. The following algorithm is executed by each searcher:

```
Algorithm Statically Allocated Parallel SCP Searcher
    Participate in parallel reductions and sorts
    Receive AdjMat, Vertex, and Set
    Build the table and list of subgraphs
    Local best cost = global best cost = ∞
    For all designated subgraphs
        Present cost = cost of next column = 0
        Loop until done
            Probe for a new global best cost
                Update local best cost if received global cost is better
            Loop while all rows not covered and
                        (present cost + cost of next column) < best cost
                Look for the next uncovered row and unused column
                If current state dominated by a previous state
                    Exit loop (Backtrack)
                End if
                If lower bound exceeded
                    Exit loop (Backtrack)
                End if
                Save column on stack, update cost, mark column and covered rows
                Probe for a new global best cost and update if required
            End loop
            If all rows covered and present cost < local best cost
                Save the columns and cost (SOLUTION)
                Send new best cost and cover to the controller
            End if
            Loop (BACKTRACK)
                Pop a column off the stack
                Remove column from solution and present cost
                Mark the rows as UNCOVERED if not covered by another column
            End loop
            If stack is empty, then search is complete
                Exit to search a new subgraph
            Else
                Advance to the next node in the search graph
            End if
        End loop
    End for
    Finished, tell controller and send performance data when polled
End Statically Allocated Parallel SCP Searcher
```

As one might expect, the static allocation of subgraphs to each searcher is effective but inefficient. Run-time testing shows many searchers idle for extended periods of time while waiting on a single searcher to finish its search. In other words, a load imbalance is occurring. The task now is to design a load balancing scheme that is more efficient than this statically allocated scheme.

*4.2.3.2 Fine Grain Algorithms* Instead of allocating the subgraphs to the searchers in advance, the parallel algorithm could assign a new subgraph to a searcher as the searcher finishes its current subgraph. This dynamic allocation scheme distributes the load more evenly between the searchers and results in an increased performance due to a decrease in searcher idle time. This fine grain algorithm moves the construction of the subgraphs from the searchers to the controller resulting in the following controller algorithm:

**Algorithm** *Fine Grain Parallel SCP Controller*
Receive **AdjMat**, **Vertex**, and **Set** from the host
Coordinate parallel reductions
Sort the **Vertex** and **Set** records
Broadcast **AdjMat**, **Vertex**, and **Set** to the searchers
Build the table and subgraphs
Send an initial subgraph to each searcher
If all subgraphs sent, tell all searchers
Loop until all searchers terminate
If a better global best cost is received
Save new global best cost and covering set
Broadcast cost to all searchers
End if
If more subgraphs exist and a searcher requests another subgraph
Send a new subgraph to the requesting searcher
If all subgraphs sent, tell all searchers
End if
End loop
Poll searchers for performance data
Send optimal covering set to host
Send performance results to host
**End** *Fine Grain Parallel SCP Controller*

This control algorithm is obviously an enhanced version of the statically allocated coarse grain control algorithm of the previous section. The controller now builds all the subgraphs in advance and sends each searcher its first subgraph. No coordination is required between the searching processors since the subgraphs are dynamically allocated

from a central processor. When each searcher has received a subgraph, the controller loops waiting for all searchers to finish, send a new global best cost, or request another subgraph. The searchers' requests for additional subgraphs are processed on a first-come, first-served basis and all searchers are notified when the list of subgraphs is depleted. Figure B.11 shows the structure chart for the fine grain controller algorithm and the ADT is given in ADT B.20.

The previous expansion algorithms are modified to build the list of subgraphs and to send each processor its initial subgraph. *Build Initial Sets* is the new expansion algorithm (ADT B.27):

**Algorithm** *Build Initial Sets*
    Put all **TABLE_NODEs** from Block 0 in the queue
    Loop until every searcher has at least one subtree to search
        Point to the first **TABLE_NODE** in the next block
        Extract a list off the queue
        Loop for all **TABLE_NODEs** in this block
            Append **TABLE_NODE** to list
            Insert new list into the queue
            Point to next **TABLE_NODE** in this block
        End loop
    End loop
    For each searcher
        Remove a subgraph from the queue
        Encode the subgraph for transmission
        Send encoded subgraph
    End loop
    Return queue to the controller
**End** *Build Initial Sets*

Notice that the steps which expand the table are essentially the same as those contained in the *Breadth-first Expansion* algorithm on page 3-13, but that additional steps have been added to provided an initial subgraph to each searching processor. Also, notice that the subgraphs are encoded for transmission. The subgraphs are composed of linked-list **TABLE_NODEs** which may not be contiguous in memory, but the iPSC/2 **csend()** command only operates on contiguous blocks of memory. Therefore, the linked-list data structure is encoded such that the information is preserved and located in contiguous memory. Figure 4.9 illustrates the encoded **TABLE_NODEs**.

When all searchers have received a subgraph, the algorithm returns the remaining subgraphs in a queue to the main controlling process. It is also worth noting that the

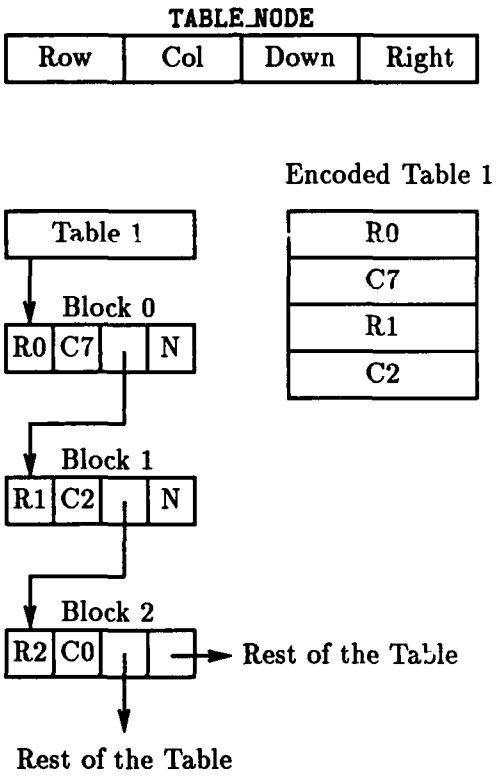



Figure 4.9. **TABLE_NODEs** Encoded for Transmission

above algorithm is easily changed to expand the subgraphs to any level (granularity) upto a complete breadth-first expansion by simply changing the first `loop`.

After the initial subgraphs are allocated, the controller waits for new global best costs or requests for new subgraphs to arrive. *Send Another Set* sends subgraphs to requesting searchers until all subgraphs are allocated (ADT B.27):

**Algorithm** *Send Another Set*
    Remove a subgraph from the queue
    Encode the subgraph for transmission
    Send encoded subgraph to the requesting processor
    Return queue to the controller
**End** *Send Another Set*

This algorithm is obviously related to the previous algorithm in the way that it sends subgraphs to the processors.

Algorithms now exist to send subgraphs to the searchers and the search algorithm must be modified to operate in conjunction with the controller. As with the controller, this fine grain search algorithm is an enhanced version of the previous statically allocated coarse grain algorithms. Since construction of the subgraphs has been moved to the controller, the searcher must now request a new subgraph whenever it finishes with its present subgraph. The new searcher is described by the structure charts given in Figures B.12 and B.8, the ADT in ADT B.23, and the *Fine Grain Parallel SCP Searcher* algorithm on the next page.

Once received, the searcher must decode the subgraph. The beginning section of the SCP table is constructed from the **TABLE_NODEs** in the received subgraph by the following algorithm (ADT B.28):

**Algorithm** *Build Table Segment*
    Loop for all rows in the encoded subgraph
        Put the row from the array into the **TABLE_NODE**
        Put the column from the array into the **TABLE_NODE**
        Point this **TABLE_NODE** to new **TABLE_NODE**
    End loop
    Point the last **TABLE_NODE** to the table block corresponding to the next row
**End** *Build Table Segment*

**Algorithm** *Fine Grain Parallel SCP Searcher*
Participate in parallel reductions and sorts
Receive **AdjMat**, **Vertex**, and **Set**
Build the table
Loop while more subgraphs available
Receive and decode a subgraph from the controller
Loop until done search of subgraph is complete
Probe for a new global best cost
Update local best cost if received global cost is better
Loop while all rows not covered and
(present cost + cost of next column) < best cost
Look for the next uncovered row and unused column
If current state dominated by a previous state
Exit loop (Backtrack)
End if
If lower bound exceeded
Exit loop (Backtrack)
End if
Save column on stack, update cost, mark column and covered rows
Probe for a new global best cost and update if required
End loop
If all rows covered and present cost < local best cost
Save the columns and cost (SOLUTION)
Send new best cost and cover to the controller
End if
Loop (BACKTRACK)
Pop a column off the stack
Remove column from solution and present cost
Mark the rows as UNCOVERED if not covered by another column
End loop
If stack is empty, then search is complete
Exit to search a new subgraph
Else
Advance to the next node in the search graph
End if
End loop
Request another subgraph from the controller and wait
End loop
Finished, tell controller and send performance data when polled
**End** *Fine Grain Parallel SCP Searcher*

In the fine grain algorithms, the subgraphs are allocated to the searchers as they finish; hence, maximum idle time is reduced. However, since NP-complete searches are inhomogeneous, it is difficult if not impossible to partition the subgraphs so that all searchers are searching for the same total length of time. Even though the individual searcher idle times are reduced, searchers are still sitting idle following depletion of the expansion queue.

The ultimate algorithm is one that has all searchers searching for the same time, that is, all searchers productively searching until the search is complete. Therefore, a scheme is required that dynamically shares *sub*subgraphs between the searchers. Then, when all subgraphs developed by the controller have been allocated and a searcher is finished, the finished searcher requests a subgraph from any searcher still searching. The dynamic load balancing algorithms of the next section accomplish this sharing of *sub*subgraphs.

*4.2.3.3 Dynamic Load Balancing Algorithms* This next version of the parallel SCP algorithm is yet another enhancement of the previous algorithm(s). The fine grain algorithm was an improvement over the statically allocated coarse grain algorithm; hence, the dynamic load balanced algorithms developed in this section add dynamic load balancing to the fine grain parallel algorithms.

The dynamic load balancing version of the SCP begins as a fine grain parallel algorithm and enters a dynamic load balancing process when all subgraphs have been distributed. Upon completion of the fine grain distribution of subgraphs, the controller triggers the active participation of a separate process called the token process. The token process exists on all processors and its only function is to coordinate the dynamic load balancing scheme. A token, Figure 4.10, is circulated through all nodes in the cube and is composed of a linear array of $m$ integers where $m$ is the number of processors in the user acquired cube. The first integer in the token (Token[0]) denotes the number of searchers still searching and is included to facilitate quick checking of the status of the search. The remaining integers in the array are used by the searchers to indicate whether they are working or idle and are set/reset by the searchers.

The token process operates differently on the controller than on the searchers; therefore, the algorithm for the token is presented in two parts. The first part follows the

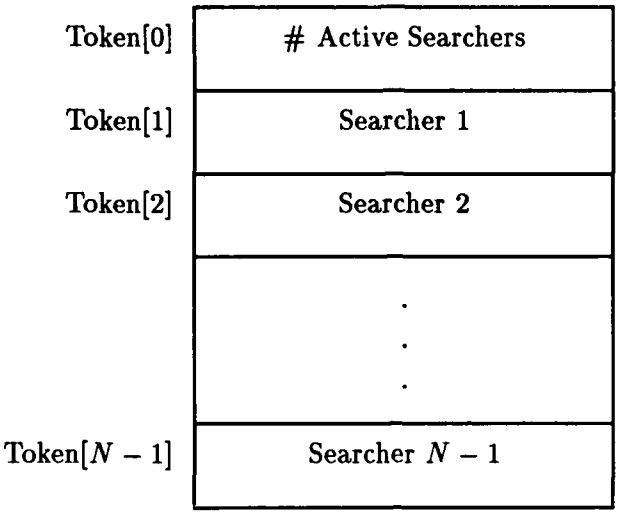



Figure 4.10. The Dynamic Load Balancing Token

controller algorithm since it resides on the control processor. The second part of the algorithm follows the searcher algorithm for obvious reasons.

The *Dynamic Load Balanced Parallel SCP Controller* algorithm is a slightly modified version of the fine grain algorithm presented in Section 4.2.3.2. The algorithm is presented on the next page, the structure chart is contained in Figure B.11, and abstract data type is represented in ADT B.19. Additional instructions are added to the *Fine Grain Parallel SCP Controller* algorithm to initiate the token process whenever all the subgraphs have been allocated to the searchers.

The *Token Controller* algorithm, also shown on next page, simply examines the first element in the token to determine if any searchers are still searching. If all searchers are waiting for another subgraph to search (Token[0] = 0), the search is complete and the token process notifies the searchers of the completion of the search. If any searcher is still executing (Token[0] $\neq$ 0), the token is passed unchanged to the next node in the ring.

The fine grain parallel searcher algorithm requires much more modification than did the controller algorithm to operate effectively in this dynamic load balanced parallel algorithm. Not only does the searcher coordinate reception of a subgraph from the token process, it must also partition its present subgraph for sharing with idle searchers. Further-

**Algorithm** *Dynamic Load Balanced Parallel SCP Controller*

```
    Receive AdjMat, Vertex, and Set from the host
    Coordinate parallel reductions
    Sort the Vertex and Set records
    Broadcast AdjMat, Vertex, and Set to the searchers
    Build the table and subgraphs
    Send an initial subgraph to each searcher
    If all subgraphs sent
        Tell searchers to start dynamic load balance
        Start the token
    End if
    Loop until all searchers terminate
        If a new global best cost is received
            Save new global best cost and covering set
            Broadcast cost to all searchers
        End if
        If more subgraphs exist and a searcher requests another subgraph
            Send a new subgraph to the requesting searcher
            If all subgraphs sent
                Tell searchers to start dynamic load balance
                Fire the TOKEN
            End if
        End if
    End loop
    Poll searchers for performance data
    Send optimal covering set to host
    Send performance results to host
```

**End** *Dynamic Load Balanced Parallel SCP Controller*

**Algorithm** *Token Controller*

```
    Start the token when notified by the controller
    While searchers still searching
        If the controller has the token
            If all searchers finished (i.e., Token[0] = 0)
                Tell searchers to stop waiting
            Else if a searcher needs another subgraph
                Send token to neighbor
            End if
        End if
    End while
```

**End** *Token Controller*

more, the searcher token process is considerably more complex than the controller token process since it must process requests for new subgraphs from its searcher as well as idle searchers contained on other processors.

Since the search algorithm is more complicated, it is divided into a *Searcher* algorithm and a *Search* algorithm. The *Searcher* algorithm obtains new subgraphs from the controller or token algorithms and passes them to the *Search* algorithm. The following algorithm is the *Searcher* algorithm. Its structure chart is illustrated in Figure B.13; while, ADT B.22 contains both the *Searcher* and *Search* algorithm ADTs.

```
Algorithm Dynamic Load Balanced Parallel SCP Searcher
    Participate in parallel reductions and sorts
    Receive AdjMat, Vertex, and Set
    Build the table
    Loop while more subgraphs available from the controller
        Receive and decode a subgraph from the controller
        Search the subgraph
    End loop
    Request a subgraph from the token process
    Loop waiting for a new subgraph from the token
        If token requests a subgraph
            Tell token I'm waiting for a subgraph
        Else if token sent a new subgraph
            Receive and decode the subgraph
        End if
    End loop
    Loop while the size of the subgraph ≠ 0
        Search the subgraph
        Request a subgraph from the token process
        Loop waiting for a new subgraph from the token
            If token requests a subgraph
                Tell token I'm waiting for a subgraph
            Else if token sent a new subgraph
                Receive and decode the subgraph
            End if
        End loop
    End loop
    Finished, tell controller and send performance data polled
End Dynamic Load Balanced Parallel SCP Searcher
```

As stated, the algorithm begins as a fine grain parallel search and enters the dynamic load balancing algorithm only when the controller has depleted its list of subgraphs. Notice that this algorithm never communicates with other searching processors. Once the load

balancing has been initiated, its only method of requesting and receiving new subgraphs is through the token process. Hence, control of the dynamic load balancing is maintained in one process (token process) which is duplicated on every processor.

The next algorithm describes the search for the optimal cover of a subgraph with dynamic load balancing and its structure chart is contained in Figure B.13.

```
Algorithm Dynamic Load Balanced Parallel SCP Search
    Loop until done
        Probe for a new global best cost
        Loop while all rows not covered and
                    (present cost + cost of next column) < best cost
            Look for the next uncovered row and unused column
            If current state dominated by a previous state
                Exit loop (Backtrack)
            End if
            If lower bound exceeded
                Exit loop (Backtrack)
            End if
            Save column on stack, update cost, mark column and covered rows
            Probe for a new global best cost and update if required
        End loop
        If all rows covered and present cost < local best cost
            Save the columns (SOLUTION)
            Send best cost and cover to the controller
        End if
        Loop (BACKTRACK)
            Pop a column off the stack
            Remove column from the solution
            Mark the rows as UNCOVERED if not covered by another column
        End loop
        If stack is empty, then search is complete
            Exit to search a new subgraph
        Else
            Advance to the next node in the search graph
            If request pending to share a subgraph
                Load balance
            End if
        End if
    End loop
End Dynamic Load Balanced Parallel SCP Search
```

The only difference between this search algorithm and the search located inside the *Fine Grain Searcher* algorithm is that a call to the following *Load Balance* algorithm is made if the token requests a subbranch for another searcher:

```
Algorithm Load Balance
    Search the stack for the highest block to be expanded
    If a block is found as SHARED
        Mark the block
        Traverse the stack to the highest block
        Send the subbranch starting at the highest block
            Encode the subbranch and send to the token process
        Return to the Search
    Else
        Tell the token process, no subbranches available
    End if
End Load Balance
```

The load balance algorithm examines the current search stack for the highest block in the table that can be expanded. In other words, it looks for the highest node in the search tree that can be expanded. Consider the search tree in Figure 4.11. The current processor is searching the subbranch starting at node 0; therefore, the subbranch starting at node 1 is the highest node in the tree that can be expanded. The load balance algorithm searches through the stack until it reaches node S. It then encodes the `TABLE_NODE` associated with node 1 and sends it to the token process. The current search algorithm then resumes its search. Realize, of course, that a certain amount of overhead is incurred by this algorithm in that the algorithm must search the stack and encode the subbranch for transmission.

The previous algorithms present the load balancing requirements levied on the searching process; whereas, the following discussion concentrates on the token algorithm. As previously stated, the token algorithm (page 4-27) resides as a separate process on all processors and passes a token around the cube. The controller's token algorithm simply examines **Token**`[0]` to determine if all searchers are finished; whereas, the token algorithm must communicate with other token algorithms and request or transfer subgraphs.

The only token process that can poll for and receive a subgraph is the token process currently holding the **Token**. A token process holding the **Token** and whose searcher is idle traverses the elements of the **Token** looking for a working processor. When found, the token process sends a request to the working processor's token process. If it receives a subgraph back from the request, it passes the subgraph to its searcher and marks its searcher as working in the **Token**. If, however, the token process polls all nodes and does

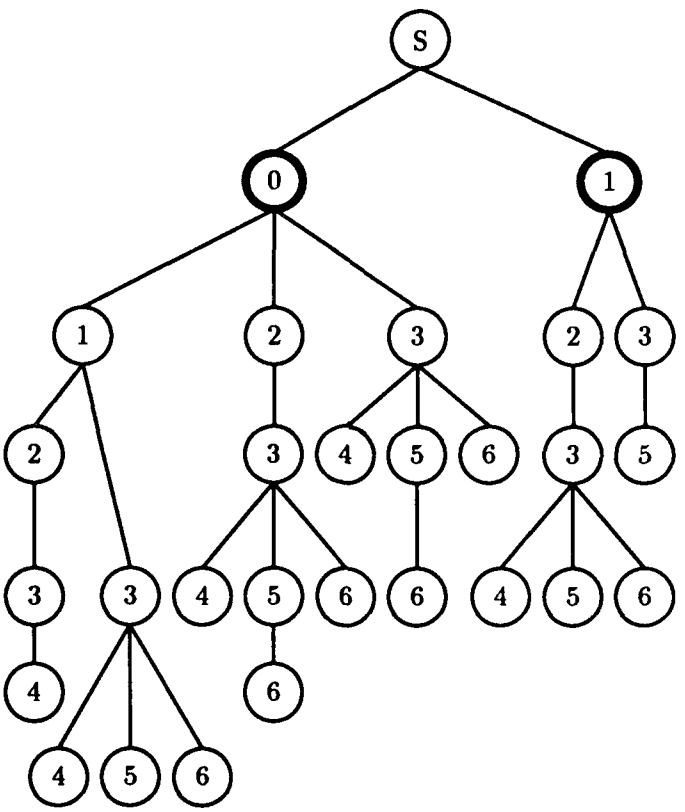



Figure 4.11. Search Tree

not receive a subbranch, it remains idle in the **Token** and the **Token** is passed to the next token process.

```
Algorithm Token Process
    Loop forever
        If my searcher wants another subgraph
            Tell controller to circulate the token
        End if
        If I have the token
            If my searcher is idle
                Mark my searcher as idle in the token
                Loop until active searcher found and not all searchers idle
                    Ask the token process of an active searcher for a subgraph
                    If that searcher is also idle
                        Mark searcher as idle in the token
                    End if
                    Look for the next active searcher
                End loop
                If active searcher found
                    Receive subgraph from token process
                    Pass the subgraph to my searcher
                    Mark my searcher as active in the token
                End if
            Else my searcher is currently active
                Pass token to token process on next processor
            End if
        Else if another token is asking for a subgraph
            If my searcher is idle
                Tell requesting token
            Else
                Ask my searcher for a subgraph
                If my searcher responds with a request
                    My searcher is now idle
                    Tell requesting token
                Else
                    Send subgraph to requesting token
                End if
            End if
        End if
    End loop
End Token Process
```

If a token process does not currently possess the **Token**, it can not request subgraphs; however, it must monitor the receive buffer for the presence of a request. If a request is present, the *requested* toker process asks its searcher for a subgraph and waits for a reply. The searcher returns a subgraph containing zero or more entries from the SCP table. The *requested* token process relays this subgraph to the *requesting* token process. A subgraph

containing more than zero SCP table entries is relayed by the *requesting* token process to the idle searcher. A subgraph containing zero SCP table entries signifies no sharing and and the *requesting* token process must poll other searching processors. As stated, the load balancing algorithms are effective but do incur a certain amount of overhead. Even so, the searchers are kept busy and the run-times indicate a decrease in the overall solution time; hence, the overhead is tolerable.

This design of the parallel SCP algorithms has progressed through three increasingly complex parallel algorithms. The first algorithm is a statically allocated coarse grain algorithm in which the searching processors expand the table and search predetermined subgraphs. The coarse grain algorithm is then modified so that the table is expanded in a central, controlling processor and the subgraphs are allocated as searching processors terminate. This fine grain algorithm performs better than the coarse grain algorithm, but run-time analysis indicates performance can be improved. Hence, a dynamic load balancing algorithm is added to the fine grain parallel search algorithms. The dynamic load balancing algorithms partition subgraphs currently being searched and share the *sub*subgraphs with other idle processors. Of the three parallel implementations, the dynamic load balancing implementation is the most efficient. The test data and results are presented in the next chapter.

The search algorithms in the previous serial and parallel versions of the SCP indicate the presence of a dominance test and a lower bound test. The details of these tests are explained in Sections E.6.6 and E.6.7 but the algorithms are not given. The following two sections present the algorithms for each test.

### *4.3 Dominance Testing*

An explanation of the dominance test is given in Section E.6.6 on page E-13. Basically, the dominance test compares the rows currently covered ($E$) plus the rows covered in the next covering column ($S_j^k$) against a list of previously saved, covered rows ($E_p$). If $E \cup S_j^k \subseteq E_p$ and $z + c_j^k \geq z_p$, the algorithm can immediately backtrack since the addition of $S_j^k$ can not result in a better cover than already obtained (17:44–45).

As the search progresses, the $E$'s are saved in a matrix, $L$, for future comparisons. Figure 4.12 represents a two dimensional, linked-list matrix indexed according to cost ($Z_i$).

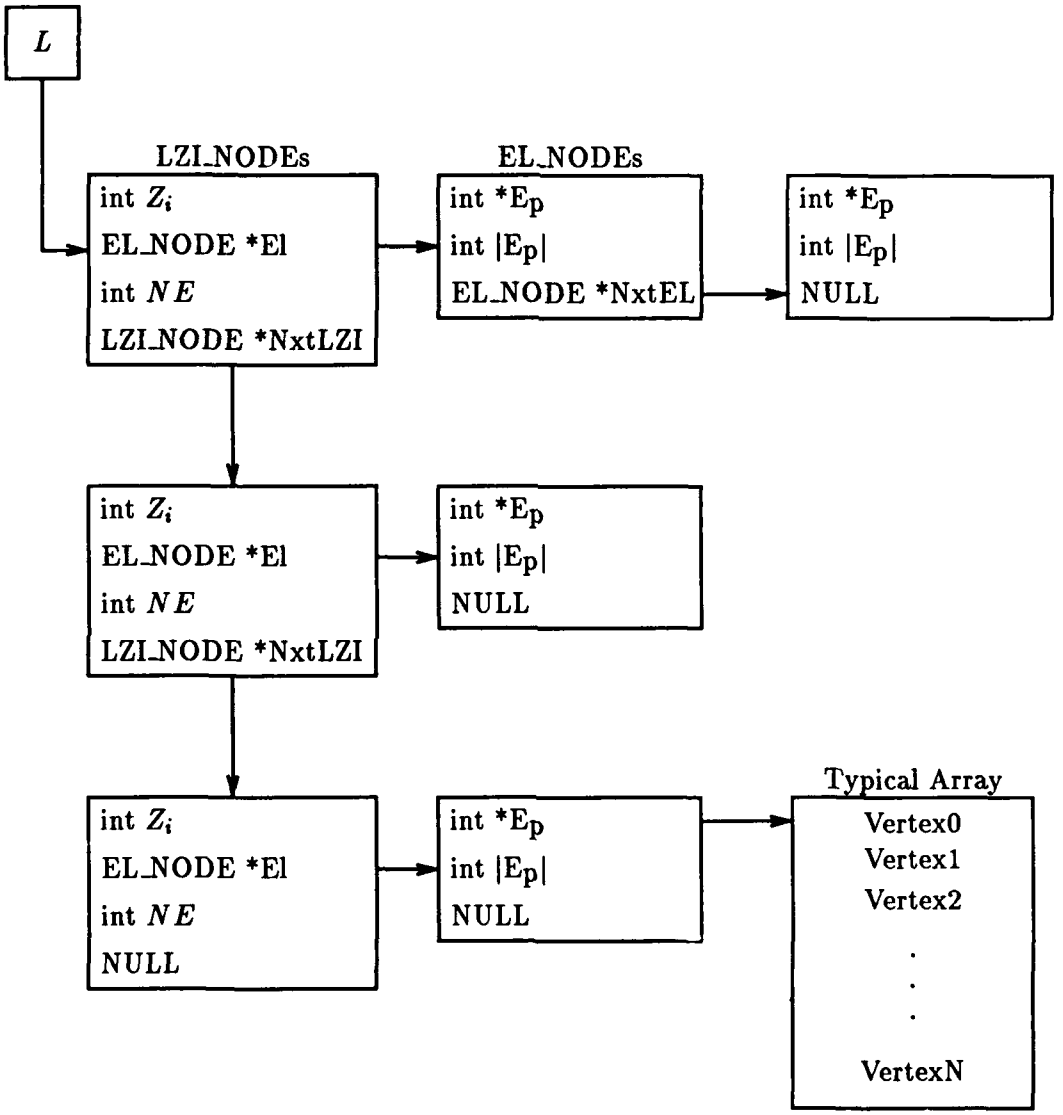



Figure 4.12. $L$ Matrix for the Dominance Test

The matrix index or rows are constructed from **LZI_NODEs** which are records containing the cost of the covered rows ($Z_i$), a pointer to an **EL_NODE** (*El), the number of **EL_NODEs** in this 'row' of the matrix ($NE$), and a pointer to the next **LZI_NODE** (*NxtLZI). The individual elements of the matrix are **EL_NODE** records containing a pointer to an array of covered

vertices (*$E_p$), the cardinality of the array ($|E_p|$), and a pointer to the next **EL_NODE** (*NxtEL). Figure 4.13 shows the C syntax for the matrix **LZI_NODEs** and **EL_NODEs**.

```
#define LZI_NODE struct lzi_node
LZI_NODE {
    int Zi;              /* Cost of all subsets in this LZI_NODE. */
    EL_NODE *El;         /* Pointer to an EL_NODE. */
    int NE;              /* Number of EL_NODEs attached to an LZI_NODE. */
    LZI_NODE *NxtLZI;    /* Pointer to the next LZI_NODE. */
};

#define EL_NODE struct el_node
EL_NODE {
    int *Ep;             /* Pointer to the set of vertices. */
    int |Ep|;            /* Number of covered vertices in Ep. */
    EL_NODE *NxtEL;      /* Pointer to the next EL_NODE. */
};
```

Figure 4.13. Language Defined $L$ Matrix Data Structures

Rather than a separate structure chart, the dominance test routines are presented as subroutines in Figure A.5. A separate abstract data type is presented as ADT A.2. The dominance test is a serial routine and all versions of the SCP (serial or parallel) are capable of executing it. Although it is possible to maintain a global list of dominating sets, the number of communications required to keep the global list current prohibits an efficient parallel implementation on a distributed computer architecture such as the iPSC/2.

The dominance test algorithm searches the $L$ matrix between the cost of the current state ($z$) and the cost of current state plus the cost of the next column chosen to add to the cover ($z + c_j^k$). If a subset is found, the algorithm returns **TRUE** signifying a dominating set exists and that the search algorithm should backtrack. Else, no dominating set is found and the algorithm should proceed forward. If no dominating set is found, $E \cup S_j^k$ is added to $L$ for use in future comparisons. If $L$ is allowed to grow without bounds, the overhead associated with searching $L$ for a dominating set becomes prohibitive and slows the overall search for an optimal covering set; therefore, the number of **EL_NODEs** allowed in any one 'row' of $L$ is constrained. Each new **EL_NODE** added to a row of $L$ is inserted into the front

of the current **EL_NODEs** and the number of elements ($NE$) is incremented. If $NE$ exceeds some predetermined threshold, the **EL_NODE** at the end of the row is dropped.

**Algorithm** *Dominance Test*
  Make the $E$ and $E \cup S_j^k$ sets
  Search $L$ between $z$ and $z + c_j^k$ for a subset of $E \cup S_j^k$
    If subset found
      Dominating set exists (BACKTRACK)
    Else if $z + c_j^k$ not present $\rightarrow$ no dominating set found (CONTINUE)
      Create a new **LZI_NODE** of cost $z + c_j^k$
      Add $E \cup S_j^k$ to $L$ at $z + c_j^k$
    Else found a matching $z + c_j^k \rightarrow$ no dominating set found (CONTINUE)
      Add $E \cup S_j^k$ at $z + c_j^k$ in $L$
      If $NE$ exceeds threshold
        Remove **EL_NODE** at end of this row
      End if
    End if
  End search
**End** *Dominance Test*

For the purposes of the complexity analysis, let $Z_{max}$ be the maximum cost row inserted into $L$ and $MAXEL$ be the threshold value for removing **EL_NODEs**. In the worst case, the entire $L$ matrix must be searched; therefore, $O(Z_{max})$ time is required to traverse the rows and $O(MAXEL)$ time is required to traverse at all elements in a row for a worst case time complexity of $O(Z_{max} \times MAXEL)$.

For the space complexity analysis, let $S_{LZI}$ represent the size of an **LZI_NODE** and $S_{EL}$ represent the size of an **EL_NODE**. The space required to contain $L$ is simply the product of $S_{LZI}$, $S_{EL}$, $MAXEL$, and $Z_{max}$ times the number of vertices stored in each array pointed to by the **EL_NODE**. Hence, the space complexity of this dominance test is $O(S_{LZI} \times S_{EL} \times MAXEL \times Z_{max} \times Nverts)$.

This completes the algorithm and complexity analysis for the dominance test. A similar development follows for the lower bound test in the next section.

### *4.4 Lower Bound Testing*

The lower bound test computes a lower bound, $h$, based on the current state ($B'$, $E'$, $z'$, $k$, $\hat{z}$) where:

$B'$ — the list of covering columns.
$E'$ — the rows covered by $B'$.
$z'$ — the cost of $B'$.
$k$ — the next column chosen to add to $B'$.
$\hat{z}$ — the cost of the best solution found so far.

The lower bound is the lowest possible cost to proceed down the path which includes $B'$ and $k$. If $z' + h \geq \hat{z}$ then the search algorithm backtracks since a better solution is not obtainable down this path.

The lower bound is calculated with a dynamic program which iterates over equation E.5:

$$g_\rho(v) = \max_{s=1,\ldots,f} [d_{\rho s} + g_{\rho-1}(v - d'_{\rho s})]$$

where "$g_\rho(v)$ is the maximum number of elements that can be covered using only the first $\rho$ rows of $D$ and whose total cost does not exceed $v$" (17:45). The construction of $D$, and a companion matrix $D'$, is described in Section E.6.7 along with a more complete explanation of the lower bound test.

The structure chart for the lower bound test routine is shown as part of Figure A.5 and ADT A.5 is the corresponding abstract data type. Since the lower bound test depends solely on local information, it is implemented as a serial algorithm and all versions of the SCP are capable of executing it. A detailed algorithm is presented followed by an order-of analysis.

**Algorithm** *Lower Bound Test*
    Make $D$ and $D'$ matrices
    Compute the lower bound, $h$
        $g_0(v) = 0$ for all $v$
        For each row in $D$ and $D'$ starting at $\rho = 0$
            $g_\rho(v) = 0$ upto the first cost in $D'(\rho)$
            Compute the next $g_\rho(v)$
                $g_\rho(v) = \max_{s=1,\ldots,f} [d_{\rho s} + g_{\rho-1}(v - d'_{\rho s})]$
        End for
        In the last $g_\rho(v)$, find $v$ such that $0 \leq v \leq \hat{z} - z$
        $h = v$, the lower bound
    If $z' + h \geq \hat{z}$
        BACKTRACK
    Else
        CONTINUE to search
    Endif
**End** *Lower Bound Test*

An analysis of the algorithm indicates that the lower bound test creates two matrices, $D$ and $D'$, for the duration of the test. In the worst case, these matrices contain $Nverts - 1$ rows and $Nverts$ columns resulting in a space complexity of $O(Nverts^2)$. On the other hand, the array containing $g_\rho(v)$ contains $\hat{z} - z$ elements. Hence, the space complexity for the lower bound test is $O(\max[Nverts^2, \hat{z} - z])$. The time complexity of the lower bound computation requires $O(Nverts^2)$ to construct the $D$ and $D'$ matrices, $O(\hat{z} - z)$ to construct $g_\rho(v)$, and $O(Nverts)$ to iterate over all rows in $D$. The time complexity is then $O(Nverts^2 + Nverts \times (\hat{z} - z))$.

The final SCP algorithms to be designed are the serial and parallel reductions in the next section. Since both serial and parallel versions of the SCP are implemented, it is necessary to design both serial and parallel reductions. Furthermore, in keeping with the design process in this research, the serial reductions are designed first and then modified to implement the parallel reductions.

### *4.5 Reduction Techniques*

This section presents the algorithms and order-of analysis for both the serial and parallel reductions. The structure chart and ADTs for the serial reductions are given in Figure A.2, ADT A.8, and ADT A.9; whereas, the structure chart and ADTs for the parallel reductions are given in Figure B.7, ADT B.11, and ADT B.16. No new data structures are required to implement the reductions and, except for Reduction #1, all reduction algorithms mark the rows and columns to be removed using the **Vertex** and **Set** vectors. Separate routines are then invoked to remove the marked rows and columns from the 0–1 matrix (**AdjMat**) and the **Vertex** and **Set** vectors.

*4.5.1 Reduction #1* Reduction #1 looks for a row with no cover and is performed while the input file is being read. The analysis of the algorithm indicates an O(1) time complexity.

*4.5.2 Reduction #2* Reduction #2 looks for all rows covered by one column. This reduction could benefit from a parallel implementation provided many rows are covered by

only one column. Such an occurrence is unlikely ; hence, a parallel version of this reduction is not designed. Thus, the following algorithm describes a serial search for rows covered by a single column:

```
Algorithm Reduction #2
    For all rows
        If the cardinality of a row is 1
            Find the column covering the row
            Save the column covering the row
            Mark the column for removal
            Mark all rows for removal which are covered by the marked column
        End if
    End for
    Remove all marked rows
    Remove all marked columns
End Reduction #2
```

The algorithm looks for all rows that are covered by one column. It saves the column for later inclusion into the optimal solution and marks it for removal. All rows that are covered by this column are also marked for removal. Separate routines then remove the rows and columns from the `Vertex` vector, the `Set` vector, and `AdjMat`.

The worst case time complexity requires a search of all rows in the `Vertex` vector to find the rows covered by one column and and then a search of all columns. If $N$ represents the number of rows and $S$ represents the number of columns, the time complexity is $O(N \times S)$. No additional data structures are created; therefore, the space complexity is $O(1)$.

*4.5.3 Reductions #3 and #4* Reductions #3 and #4 remove dominated rows and columns; hence, a search is required to find and mark the dominated rows and columns. Given that the entire matrix is searched in the process of finding dominated rows or columns, these two reductions could benefit from a parallel implementation of the search. The search algorithms for both reductions are closely related; therefore, only Reduction #3 is presented.

The following algorithm is a serial version of Reduction #3 from which a parallel version is later derive:

```
Algorithm Serial Reduction #3
    Start at the first row (ThisRow)
    NextRow = ThisRow + 1
    For all rows in Vertex
        If cardinality of NextRow ≤ cardinality of ThisRow
            If NextRow ⊆ ThisRow
                Mark ThisRow for removal
                Select another row for ThisRow
            Else
                Select another row for NextRow
        Else the cardinality of NextRow > cardinality of ThisRow
            If ThisRow ⊆ NextRow
                Mark NextRow for removal
            Select another row for NextRow
        End if
    End for
    Remove marked rows
    Renumber Index fields in the Vertex vector
End Reduction #3
```

The algorithm continually selects unmarked rows for comparison with other unmarked rows. If 'ThisRow' is dominated by 'NextRow', then 'ThisRow' is marked for removal and the pointer set to the next unmarked row. If 'NextRow' is dominated by 'ThisRow', then 'NextRow' is marked for removal and the pointer set to the next unmarked row. In this manner, the algorithm compares all rows. When all dominated rows are marked, a separate routine removes the marked rows and renumbers the **Index** fields of the remaining records in the **Vertex** vector. The removal algorithm removes rows in the matrix by moving blocks of memory. In other words, a row $R$ is removed by shifting the following rows, $R+1 \rightarrow R+N$, into the memory location previously occupied by $R$.

The search for the dominated rows requires

$$\frac{N(N-1)}{2}$$

comparisons and the removal of the vertices requires

$$N + SN_r$$

memory moves where $N_r$ represents the number of vertices to be removed. The order-of is then

$$\frac{N(N-1)}{2} + N + SN_r$$

The parallel reduction algorithm is a modified version of the serial algorithm. In the following algorithm, MyNID is a node processor's identification number.

```
Algorithm Parallel Reduction #3
    Give each node the matrix
    Divide the vertices equally among all nodes
    For each node
        Receive a list of vertices to search
        Find the dominated vertices (serial search)
        Iterate cube dimension times
            If MyNID ∧ 2 Iteration ≠ 0
                Send list of vertices to node MyNID ⊕ 2 Iteration
                Abort iteration
            Else
                Receive list from node MyNID ∨ 2 Iteration
                Find the dominated vertices (modified serial search)
            End if
        End iteration
    End for
    Remove marked vertices
    Renumber index field in the vertex record
End Reduction #3
```

The controller sends all processors an equal portion of the **Vertex** vector and keeps a portion for itself. The initial vector is searched for dominated rows using the serial reduction algorithm previously presented. When a processor has completed its search, it passes or receives a list of vertices to/from a neighbor. For example, in a cube consisting of eight nodes, node 1 sends to node 0 ($1 \rightarrow 0$), $3 \rightarrow 2$, $5 \rightarrow 4$, $7 \rightarrow 6$. Nodes 0, 2, 4, and 6 then execute the search algorithm again and send their lists in the following manner: $6 \rightarrow 2$, $4 \rightarrow 0$. Finally, nodes 0 and 2 execute the search again and combine in the following manner: $2 \rightarrow 0$. The process is executed $d$ times where $2^d$ is the number of nodes in the cube. The marked copy of the entire **Vertex** vector is now in node 0 where the rows are removed. The advantages of this parallel algorithm is that at each iteration, the node processor only compares unmarked rows on $2^d$ equal parts of the **Vertex** vector are being examined simultaneously.

The complexity analysis follows from the previous analysis. Each of the $2^d$ processors receives an equal portion of the **Vertex** vector; hence, each processor receives $\frac{N}{2^d}$ elements.

The initial comparison of the rows requires

$$\frac{\frac{N}{2^d}(\frac{N}{2^d}-1)}{2}$$

comparisons. The second iteration requires

$$\left(\frac{N}{2^d}\right)^2$$

comparisons and the $d^{\text{th}}$ iteration requires

$$\left[2^{(d-2)}\left(\frac{N}{2^d}\right)\right]^2 = \frac{N^2}{16}$$

comparisons. The removal of the vertices is still accomplished by one processor and is the same as in the analysis of the serial reduction. Therefore, the time complexity for the parallel reduction is

$$\frac{N^2}{16} + N + SN_r$$

This completes the design and analysis of the reduction algorithms. The last section explains the parallel sorting algorithm implemented for this research.

### *4.6 Parallel Bitonic Merge Sort*

Sorting is a common activity in many algorithms and the SCP algorithms are no exception where the **Vertex** and **Set** vectors are sorted to improve the efficiency of the search algorithm. Many sorting algorithms have been designed and implemented on parallel computers; hence, parallel sort algorithms are easily obtained (16, 25, 50). The parallel sort employed in the SCP is an enhanced version of a bitonic merge sort obtained from Quinn (50:93–94). This version of a bitonic merge assumes $2^d$ items are sorted in ascending order on $2^d$ processors. The bitonic merge designed for the parallel SCP algorithms sorts any size input data in either ascending or descending order on $2^d$ processors.

There are two algorithms required to implement the parallel sort. The controller algorithm performs a check on the number of items to be sorted and tells the rest of the

processors the type of sort to be performed (serial or parallel). The serial sort is a quick sort and is included to improve the overall efficiency of the sorting package. The quick sort is executed if the number of items is less than some empirically determined threshold. In the case of a parallel sort, the controller partitions the data such that every processor, including the controller, is assigned a near equal portion of the data. If the number of items is divisible by $2^d$, then all processors receive an equal number of items; otherwise, the lower addressed processors receive one additional item and the other processors pad their data with a user supplied pad value.

When all processors have received their list of items, a parallel bitonic merge sort is executed synchronously between $2^d$ processors. After $d$ iterations, each processor contains a portion of the sorted list of items. The controller algorithm now collects the lists from the other processors and removes any padding according to a user supplied function.

The structure charts for these two algorithms are contained in Figures B.6 and A.3, and the abstract data type in ADT B.10. For the following control algorithm, let $N$ represent the number of elements to be sorted, MyNID represent a node processor's identification number ($0 \leq \text{MyNID} \leq 2^d - 1$), and $d$ represent the dimension of the cube.

```
Algorithm Parallel Sort Controller
    If N < THRESHOLD and MyNID == CONTROLLER
        Do a quick sort and tell the other processors
    Else
        If MyNID == CONTROLLER
            Tell the other nodes to do a parallel sort
            Divide items between the processors
            Participate in a parallel bitonic merge sort
            Receive sorted sublists from the other processors
            Remove padding if input list not divisible by 2^d
            Return sorted items
        Else
            Get a portion of the input list
            Pad list if received portion not divisible by 2^d
            Participate in a parallel bitonic merge sort
            Send the sorted sublist to the CONTROLLER
            Return
        End if
    End if
End Parallel Sort Controller
```

The above algorithm controls the actual bitonic merge sort described in the following algorithm:

```
Algorithm Parallel Bitonic Merge Sort
    Sort the initial list (quick sort)
    For i = 0 to d − 1 do
        For t = i downto 0 do
            d = 2^t
            If (MyNID mod 2 × d) < d then
                Receive a new bArray from node MyNID + d
                Merge aArray and bArray (merge sort)
                if (MyNID mod 2^(i+2) < 2^(i+1)) and sort DESCENDING
                    bArray is the max half of the list
                    aArray is the min half of the list
                Else if (MyNID mod 2^(i+2) >= 2^(i+1)) and sort ASCENDING
                    bArray is the min half of the list
                    aArray is the max half of the list
                End if
                Send bArray to node MyNID + d
            Else
                Send aArray to node MyNID − d
                Receive bArray from node MyNID − d
            End if
        End for
    End loop
End Parallel Bitonic Merge Sort
```

Rather than explain the process of the bitonic merge sort, the reader is referred to Quinn (50:81–106). If $N$ is the number of items to be sorted and is divisible by $2^d$, each processor receives $N/2^d$ elements. For a cube of dimension $d$, the inner instructions execute

$$\frac{d(d+1)}{2}$$

times and the merge sort moves

$$\frac{2N}{2^d}$$

items during each iteration. Since the algorithm is synchronous, all nodes must wait for the merge sorts to complete. Hence,

$$\left(\frac{2N}{2^d}\right)\left(\frac{d(d+1)}{2}\right) = \frac{Nd(d+1)}{2^d}$$

items are moved during the algorithm. Additional space is required during execution to maintain two arrays ($aArray, bArray$) of size $\frac{N}{2^d} + 1$; therefore, the space complexity is $O(\frac{N}{2^d})$.

## *4.7 Summary*

This chapter has presented detailed designs and analysis for a serial and three parallel SCP algorithms in Sections 4.2.2, 4.2.3.1, 4.2.3.2, and 4.2.3.3. Furthermore, algorithms for the SCP dominance test, the SCP lower bound test, the SCP reductions, and a parallel bitonic merge sort are presented in Sections 4.3, 4.4, 4.5, and 4.6. The design process throughout has been to develop a simple serial algorithm and then conduct a parallel decomposition to derive a parallel algorithm.

The serial SCP algorithm is presented first. It is a slightly enhanced version of the branch-and-bound algorithm presented by Christofides (17:41–42). A parallel decomposition of the serial algorithm is accomplished to develop the parallel algorithms. A common control structure is employed for all parallel SCP algorithms; namely, a controller acts as a central memory where the searching processors submit their best cost cover. The controller evaluates the new covers, keeps the best, and broadcasts a global best cost to all searchers which then use the global best cost to further prune their search trees.

A parallel version of the serial algorithm is presented based on a coarse grain decomposition of the search tree. In the coarse grain algorithm, the searching processors do most of the work since they must build the initial subgraphs and then select predetermined subgraphs to search. Run-time analysis reveals that many of the processors are idle during much of the search; therefore, a better method of load balancing is developed.

The coarse grain algorithm is modified by placing the building of the subgraphs at the controller and then partitioning subgraphs to the searching processors. When the searchers finish their subgraph, they request new subgraphs from the controller. When the controller's list of subgraphs is depleted, the searching processors remain idle until all searchers have completed. Again, run-time analysis reveals a load imbalance though not as great as with the coarse grain algorithm. Since a load imbalance still exists, a dynamic

load balancing algorithm is designed.

The dynamic load balancing algorithm adds a dynamic load balancing process to the previous fine grain algorithm. Now, when the controller depletes its list of subgraphs, it triggers a token which starts the dynamic load balancing process. The token traverses the cube through all processors. If a searcher is idle, it informs its dynamic load balancing (DLB) process and waits for a response. The DLB process waits for the token to arrive and then checks the token for working processors. The working processors are polled until all refuse to send a subgraph or a subgraph is received by the requesting DLB process. The received subgraph is sent to the requesting searcher and the token is passed to the next processor in the ring. Each time the controller receives the token, it checks the number of processors still working. If all processors are idle, the search is complete. This final parallel version of the SCP proves to be the most efficient according to the run-time analysis.

Following the design of the parallel SCP algorithms, algorithms for the dominance test, lower bound test, and parallel reductions are presented. The dominance test and lower bound test algorithms are serial. The dominance test is serial because the huge volume of communications required to keep global $L$ matrix up to date would quickly swamp the interprocessor network. The lower bound test is serial because it depends on local information. Serial algorithms for Reduction #1 and 2 are presented along with a parallel algorithm for Reduction #3.

The final section presents an algorithm and analysis for a parallel bitonic merge sort. Since sorting is an integral part of the SCP algorithms and parallel computers are efficient sorters, a parallel sort increases the overall efficiency of the parallel SCP search algorithms.

This completes the design and analysis of the algorithms for a parallel implementation of the SCP started in Chapter III. The next chapter discusses the performance metrics, the test cases, and the test results obtained from executing the algorithms designed in this chapter.

# V. Results

## 5.1 Introduction

The previous two chapters present the design of the serial and parallel SCP programs. The design of the parallel programs led to three increasingly complex algorithms: coarse grain, fine grain, and dynamic load balanced. The coarse grain algorithm is developed first and tested to assess its performance. The tests show many processors idle during much of the search. This idle-time is unacceptable for this research; hence, the coarse grain algorithm is modified and a fine grain algorithm is developed. The fine grain algorithm exhibits less overall idle-time; hence, it is more efficient than the coarse grain version. However, the maximum idle-time (the difference between the first terminating searcher and last terminating searcher) is still measured in minutes; hence, the dynamic load balanced algorithm is developed and tested. The dynamic load balanced algorithm adds a dynamic load balancing or sharing to the fine grain algorithm. The result is an efficient parallel SCP implementation with individual processor idle-time measured in seconds.

The development of the three parallel algorithms is based on idle-time observed during execution. The purpose of this chapter is to discuss the data gathered during execution of the algorithms. Section 5.2 discusses the performance metrics gathered during the run-time testing of the parallel algorithms. The test plan and input test cases are discussed along with a discussion of the program used to generate the test cases, Section 5.3. The results of the test are presented in Section 5.4 with the tabulated results contained in Appendix D. This chapter does not interpret the results, it simply explains how they were generated and how to read the tables. The interpretation is left to Chapter VI.

## 5.2 Performance Metrics

The purpose of this section is to define a set of performance metrics for evaluating the efficiency and effectiveness of the SCP algorithms. The selection of a 'good' set of performance metrics is difficult because the collection of too much data could adversely affect the program's efficiency and, if too little data is collected, an interesting observation might be missed.

A list of general performance metrics are presented on page 3-3 of Chapter III and are expanded here to address their use in the SCP programs.

**Total Program Execution Time** — Total execution time for the SCP program. The total program execution time is of interest since this measure includes all processing and communications time.

**Search Time** — Time the algorithms actually spent searching. Does not include time to build the table, sort the data, and so forth. The majority of this time is expanding the search tree; although, the time to send and receive global best costs is also included.

**Expanded Nodes** — Number of nodes expanded in the search tree by each searching processor. In the dynamic load balanced version of the SCP, this metric includes the additional nodes a processor expands because it received a subgraph from another processor. This measure is a useful indication of the extent to which the search tree was expanded.

**Processor Idle Time** — Time the searching processors spend waiting for data to search. In general, the most efficient search occurs when all processors are equally productive. In other words, each processor is productively searching various subgraphs. Therefore, this metric is useful in determining the relative productivity of each processor.

**Minimum Solution Time** — Program execution time until the optimal solution was first found. As the processors search, they submit proposed optimal solutions to the controller. The faster the optimal solution is found, the faster it can be used to bound the search tree on all processors. In the SCP programs, this metric is identified as the *time until the best cover was first found.*

**Global Best Cost Broadcasts** — Number of times the global best cost is broadcast to the searching processors. This metric is a relative measure of the communications traffic between the controller and the searching processors. It could indicate potential communications bottlenecks.

**Support Time** — Time to sort the input data, build the table, and execute any reductions. Many of the support routines are parallel; hence, these times allow a

comparison between the serial and parallel algorithms. The SCP programs report these times separately.

**Search Efficiency** — Sum of the individual processor search times divided by the product of the total program execution time and the number of searching processors. This metric is a percentage representation of the overall processor search efficiency.

**Speedup** — Execution time of the best serial algorithm divided by the execution time of the parallel algorithm. This metric is a standard by which many parallel algorithms are judged. It represents whether a parallel implementation is taking full advantage of the additional processing power available in a parallel architecture.

**Load Balance Time** — Total time each processor spent developing subgraphs to share with idle processors. Although the load balancing algorithm is important, if it requires too much time to partition the subgraph, any potential speedup could be lost.

**Search State** — The state of the search at each node in the search tree. This metric is useful in studying the expansion of the search tree on each searching processor. Since the memory on the node processors is limited, this metric is not implemented in the current algorithms, reference Section 6.5.

This list of metrics is sufficient to measure the efficiency and effectiveness of the SCP algorithms. Other metrics are available, such as the number of search tree nodes shared or passed across the network, number of times the dynamic load balancing token traversed the cube. However, it is pointless to collect such metrics unless they contribute to the interpretation and use of the programs. All of the above metrics, excluding the **Search State**, are available in the SCP algorithms. Based on these metrics and the available SCP program options, reference Appendix C, a test plan is developed to test the serial and parallel SCP programs.

### *5.3 Test Plan*

Figures 5.1 and 5.2 show the help screens available by typing **scp** on the command line (ref. Appendix C for an explanation of the various command line arguments). The

test plan validates the correct operation of each user selectable option in isolation and in concert with other options. Following the test of the options, the efficiency and effectiveness testing of the search algorithm in the two serial and three parallel programs is discussed.

```
        SERIAL SET COVERING PROBLEM (SCP) HELP SCREEN

>scp [r234dlats] File

Options available:
  r - Enable all reductions.
  2 - Enable reduction #2.
  3 - Enable reduction #3.
  4 - Enable reduction #4.
  d - Enable dominance testing.
  l - Enable lower bound testing.
  a - Print the 0-1 matrix.
  t - Print the table.
  s - Save the reduced/reordered 0-1 matrix to a file.
```

Figure 5.1. Serial SCP Help Screen

The effectiveness of the various algorithms is judged mostly be comparison. That is, a set of test problems are searched with all versions of the SCP and the results compared. Certainly not a proof that the algorithms work, but a good indicator. As discussed in Section 3.5.2, the serial SCP routines are developed on a personal computer using Turbo C which includes an integrated debugger. Turbo C's debugger is used to monitor the operation of the serial routines; hence, the serial version of the SCP and its routines were closely observed in a glassbox environment. Furthermore, the serial version of the SCP is the simplest version developed. For these two reasons, glassbox monitoring and simple search, the solutions derived from the serial SCP are considered correct.

Eluded to in the previous paragraph is the existence of a set of test matrices. Twenty-four such matrices are generated with a matrix generation algorithm called **gentable**. **Gentable** prompts for the number of rows and columns, the density of the matrix, and the range of the costs. The 1's and 0's are randomly generated according to the specified density where the density is defined as the number of 1's divided by the total number of

```
  PARALLEL SET COVERING PROBLEM (SCP) HELP SCREEN

>scp [Algorithm|Options] Searchers File

Algorithm|Options:
  C - Runs coarse grain SCP program -- GOOD.
  F - Runs fine grain SCP program -- BETTER.
  D - Runs dynamic load balanced SCP program -- BEST (default).

  S - Force a serial sort in the parallel search.
  P - Force a parallel sort in the parallel search.
  r - Enable all reductions.
  2 - Enable reduction #2.
  3 - Enable reduction #3.
  4 - Enable reduction #4.
  d - Enable dominance testing.
  l - Enable lower bound testing.
  n - Print individual node statistics.
  a - Print the 0-1 matrix.
  t - Print the table.
  s - Save the reduced/reordered 0-1 matrix to a file.

Searchers:
  Number of searching processors = 0---7(default).
  0 - Serial SCP program on one node.
```

Figure 5.2. Parallel SCP Help Screen

matrix elements. Following generation of the 0–1 matrix, the costs are randomly generated between the user specified range. These matrices are searched with the serial SCP program and the optimal solutions tabulated in Table D.1. The table contains a total of twenty-nine matrices since matrix **2020** has four versions which were generated by **gentable** and then modified to test a particular characteristic of the algorithms. For instance, **2020v2** has two rows which are covered by only one column. This particular matrix tests the effectiveness of reduction algorithm #2 and the search algorithm in the absence of the reduction.

As stated previously, these test matrices represent examples in which all versions of the SCP performed effectively. This does not prove that the algorithms will perform correctly in all instances; however, since all algorithms return the same optimal cost for all test cases, the algorithms are assumed to be correct. The testing of the program options is presented in the following sections.

*Reduction Testing* The effectiveness of the serial SCP reductions is checked by inserting known problems into the algorithm and then observing the reduced output. The known problems contain an example problem from Christofides (17:54), a matrix containing rows covered by only one column, a matrix containing columns with only one element, a matrix containing all 1's on the diagonal, and a unit cost matrix. The serial SCP program returns the correct reduced matrix for the known problems; therefore, the remainder of the test matrices (solutions unknown) are reduced. The results from the serial SCP are considered correct and are compared against the results from the parallel algorithms. All versions of the parallel programs execute the same reduction algorithms; therefore, only one parallel SCP algorithm is tested to compare its reduced matrices against the serial reduced matrices. The results indicate that both the serial and parallel reduction algorithms produce the same reduced matrix.

The efficiency of the reductions refers to the serial reduction time versus the parallel reduction time. The serial SCP algorithms are executed with the reductions enabled on a large (100x100) matrix. The same input matrix is tested with the parallel SCP programs and the results are reported in Table D.2. The table shows the time to reduce the matrix with the serial reductions and with parallel reductions executed on three different size

cubes.

*Bounding Function Testing* The bounding functions are serial algorithms; therefore, their effectiveness is judged with the serial SCP algorithm. It is much too time consuming to expand a search tree by hand and then reduce the search tree using the dominance test and lower bound tests; therefore, these two routines are checked via a confidence test. The search is executed with and without the bounding tests and the number of nodes expanded in the search tree is observed. Assuming that the bounding tests are effective, one would expect to see less nodes expanded with either or both of the bounding tests enabled. Furthermore, the serial algorithm should return the same optimal covering sets as well as the same optimal cost. The same tests are executed with the parallel algorithms. The efficiency of the tests is easy to observe given the `Search Time` performance metric described in Section 5.2.

*Display Routine Testing* The effectiveness of the print routines and the save matrix routine are easy to observe. The displays and saved files should correlate with the input problems. The efficiency is not relevant since the displays are simply informational.

*Sort Routine Testing* The effectiveness of the sorting algorithms is checked by displaying the matrix after it is sorted. A measure of their efficiency is obtained from the sort time performance metric given by the SCP algorithms upon completion of the search. Table D.2 shows the time to sort the rows and columns with the serial and parallel algorithms.

*Search Algorithm Testing* The effectiveness of the three parallel search algorithms is judged by comparing the returned optimal cost for the test matrices against the optimal cost obtained from the serial SCP algorithm. The cost of the serial and parallel algorithms must be equal, even though the covering sets may be different. Furthermore, the parallel algorithms must return the same optimal cost for any number of searching processors. All parallel algorithms do indeed return the same optimal cost per problem, Table D.1, for 1–31 searching processors.

A major focus of this research is the efficiency of the parallel search algorithms; hence, the question of how to measure the search efficiency is of great concern. A common measure of parallel program performance is the speedup defined by (56:93):

$$S(N) = \frac{t_{serial}}{t(N)}$$

Although speedup indicates the relative speed of the parallel versus serial algorithm, it does not allow an interpretation of the efficiency of the parallel programs. For instance, are all processors working all the time and are they contributing to the solution or just executing a redundant search? Additional performance metrics are required to answer these questions.

The SCP programs provide more information than just the execution times required to compute the speedup. They also show the time that the best cost was last updated, the time spent searching, the time spent waiting for subgraphs to search, and the number of nodes expanded. The time spent searching versus the overall execution time and the time spent waiting for subgraphs to search are measures of the processor idle-time. The total execution time refers to the time required to find the optimal solution. In other words, it is the time to completely search the input matrix. Searching processors that do not productively search the entire length of the total execution time are wasted. Hence, considerable effort is exerted in this research to minimize the idle-time. In fact, the minimization of processor idle-time is the prime motivation for developing the three versions of the parallel algorithms. The maximum idle-time is tabulated in Table D.3 and is defined as the amount of time between when the first searching processor stopped searching and when the search was complete.

The SCP program output is not included in this document; however, it is available in the `/results` directory of the SCP files and an example is shown in Section C.5. The results of the testing are tabulated in Appendix D with an explanation of the tables given in the next section.

## *5.4 Test Results*

The purpose of this section is to explain how to read the tables with the interpretation of the tables left for the next chapter. As shown in Table 3.1, AFIT's iPSC/2 computer contains only eight nodes. To gather more data, customer service representatives at Intel were contacted and an account on a 64-node iPSC/2 computer was obtained. Much of the results presented in the tables were gathered on their computer. The following descriptions indicate which data is obtained from AFIT's 8-node hypercube and which is obtained from Intel's 64-node hypercube.

**Table D.1** — Shows the optimal solution and a corresponding list of covering sets to the twenty-nine test problems. These solutions were obtained from the serial SCP algorithm executed on AFIT's hypercube.

**Table D.2** — Execution times for the sorting and reduction algorithms. The serial times are the algorithms executed by the serial SCP program and the parallel times are from the dynamic load balanced parallel SCP algorithm executed on three different cube dimensions since the parallel sort and reduction algorithms require $2^d$ processors where $d$ is the dimension of the cube. The serial sort algorithm is a quick sort and the parallel sort is a bitonic merge sort. These times were collected at AFIT.

**Table D.3** — The maximum time that a searching processor was idle during a search of the test matrix. This time is the maximum obtained for any number of processors. The data is obtained from Intel's hypercube.

**Table D.4** — A serial SCP program is available which executes on a node processor rather than the host processor. This program is a strict serial program and it contains no parallel algorithms. It is executed on a node processor because the node processors are single user; hence, all CPU time is dedicated to the single user and the program finishes sooner. This table shows the elapsed time from the beginning of the search until the best cost was last updated (`BCT`), the search time (`ST`), the total execution time (`TT`), and the number of search tree nodes expanded (`EN`). Only the solution times for the five most time consuming test matrices is shown since the rest of the matrices are solved much too quickly for use in comparing the serial and

parallel search algorithms. These times are the base times from which the speedup is calculated were obtained from Intel's hypercube.

**Tables D.5, D.6, and D.7** — These tables contain the same four measurements shown in Table D.4 (i.e., **BCT**, **ST**, **TT**, and **EN**) for the same five test matrices searched on Intel's hypercube. The **searchers** column refers to the number of processors searching the input matrix and ranges from 1 to 31 searching processors.

**Table D.8** — The speedup obtained when searching one of the five test matrices for 1–31 searching processors. The speedups are computed by dividing the times displayed in Table D.4 by the search times (**ST**) displayed in Tables D.5, D.6, and D.7.

### *5.5 Summary*

This chapter has discussed the performance metrics developed to test the efficiency and effectiveness of the serial and parallel SCP programs. Not all possible metrics are collected and analyzed. The effectiveness of the programs is judged mostly by comparison. The serial SCP program is simpler than the parallel versions and is observed extensively in a glassbox environment. Furthermore, the serial version of the SCP returns the correct result for known test problems; hence, the solutions obtained from the serial SCP are assumed correct and are used to validate the correctness of the solutions returned by the parallel algorithms. The test matrix generation program is discussed along with the test plan for the SCP options. A significant portion of this research is dedicated to improving the efficiency of the parallel SCP programs; hence, a test plan to measure these improvements is presented. The results of the testing are tabulated and presented in Appendix D. The last section of this chapter, Section 5.4, explains the tables and the data contained in them. The interpretation of this tabulated data is the subject of the next chapter.

# *VI. Conclusions and Recommendations*

## *6.1 Introduction*

Chapters I and II describe the nature of NP-complete problems and present existing methods for designing parallel search algorithms. Furthermore, Chapter I introduces the set covering problem (SCP) and supportive reasons for its selection as a representative of the class of NP-complete problems. Chapter II describes the SCP in more detail; however, a complete description of the problem is left to Appendix E. For reference purposes, Appendix E also contains descriptions of four other NP-complete problems; namely, the assignment problem, the hamiltonian circuit problem, the traveling salesman problem, and the 0/1 knapsack problem.

The preliminary design as evolved in Chapter III focuses on the development of a parallel SCP algorithm with the detailed design presented in Chapter IV. The preliminary design development introduces a data parallelism approach to parallelizing the SCP, as well as, the initial data and control structures. Formal definitions of serial and parallel branch-and-bound SCP algorithms are presented in the context of UNITY metaprograms for the high-level design.

The parallel branch-and-bound UNITY metaprogram is mapped to an architecture specific UNITY metaprogram in Chapter IV. The remainder of Chapter IV presents pseudo-code algorithms for the serial and parallel versions of the SCP. Three different parallel versions of the SCP are developed based on a desire to decrease individual processor idle-time and are directed at improving the load balancing characteristics of the parallel algorithms.

Chapter V describes the performance metrics employed to gather information about the programs. It also describes the test plan for the algorithm evaluations and explains the raw results tabulated in Appendix D.

An interpretation of the results is the purpose of this chapter, Chapter VI. Section 6.2 is an interpretation of the results reported in Appendix D. This section justifies and explains the results and relates the results to the complexity analysis conducted in

Chapter IV. Section 6.3 contains concluding remarks concerning the relevance of this research and compares the results with those published by other authors. Section 6.4 discusses major problems encountered during this research. It is included so as to help future users avoid some of the pitfalls experienced in parallelizing NP-complete problems on a cube-connected computer. The last section, Section 6.5, lists recommendations for further research.

## *6.2 Interpretation of the Results*

Before the actual search execution of the specific input problem is initiated, the data is sorted and any user requested reductions are performed. As presented in Chapter IV, a parallel bitonic merge sort and three parallel reductions are available. The results of the testing are presented in Table D.2.

*6.2.1 Sorting* It is difficult to compare the efficiency of the bitonic merge sort with the quick sort available in the serial SCP since the maximum size of the input matrix is constrained by the amount of memory available on the node processors (per Table 3.1). Nevertheless, a $1000 \times 700$ matrix is generated and the rows and columns are sorted. The time required to perform a quick sort on both the rows and columns, as shown under the 'Serial' column in Table D.2, is 0.49 seconds. The time required required to perform a bitonic merge sort is shown under the 'Parallel' column for three different cube dimensions and ranges from 0.22 to 0.35 seconds. Even for this small input problem, the bitonic merge sort out performs the quick sort.

The analysis of the bitonic merge sort in Section 4.6 gives the order-of as

$$\frac{Nd(d+1)}{2^d}$$

moved items where $N$ is the number of items to be sorted and $d$ is the dimension of the cube. The quick sort is typically cited as $O(n \log n)$ (10:462). Given the memory constraints, the bitonic merge sort is validated by example. In other words, test input matrices are sorted and the results are visually inspected. However, the data in the table

indicates that the parallel sort is more efficient. Since the study of the bitonic merge is not the objective of this research, no further work or analysis is performed.

*6.2.2 Reductions* Table D.2 also contains the time to execute all reductions for the serial and parallel algorithms. Since the test matrices listed in Table D.1 are reducible in such a short time, the 1000 x 700 matrix generated for the sort routines is used and the SCP algorithms are modified to execute only the reduction algorithms (i.e., no search). The results indicate that the parallel algorithms are about 1.5 times faster than the serial reduction algorithms. As with a comparison of the sorting algorithms, the size of the input data is limited by the node processor's main memory. Since large input problems can not be tested without major modifications to the SCP software, no further testing is accomplished. The only empirical conclusions possible for such small problems is that the parallel reductions are faster than the serial reductions.

*6.2.3 Searching* In contrast to the sorting and reduction algorithms, extensive testing was conducted on the parallel search algorithms. Data is gathered from each one of the controller and searching processors. The raw data is not included in this document due to the volume, but is available in the `/results` directory containing the SCP programs.

Much of the justification for implementing three different parallel versions of the SCP is based on reducing individual processor idle-time. Table D.3 shows the maximum idle-time obtained when the three parallel SCP algorithms searched the five test matrices. The results indicate that, in general, the coarse grain algorithm incurred the most idle-time (85–1065 seconds) followed by the fine grain algorithm (152–144 seconds) and then the dynamic load balanced (DLB) algorithm (19–42 seconds); however, these numbers are deceiving. If one were to judge the parallel algorithms based solely on processor idle-time, the DLB algorithm is clearly the most efficient algorithm and the fine grain algorithm is usually better than the coarse grain algorithm. Such a conclusion is invalid as shown in the following discussion.

Tables D.4, D.5, D.6, and D.7 document the best cover time (BCT), the search time (ST), the total time (TT), and the number of expanded nodes (EN) for the serial and

three parallel SCP algorithms. A normalized speedup is calculated by dividing the search time contained in Table D.4 by the parallel search times. These normalized speedups are contained in Table D.8 and plotted in Figures 6.1, 6.2, 6.3, 6.4, and 6.5.

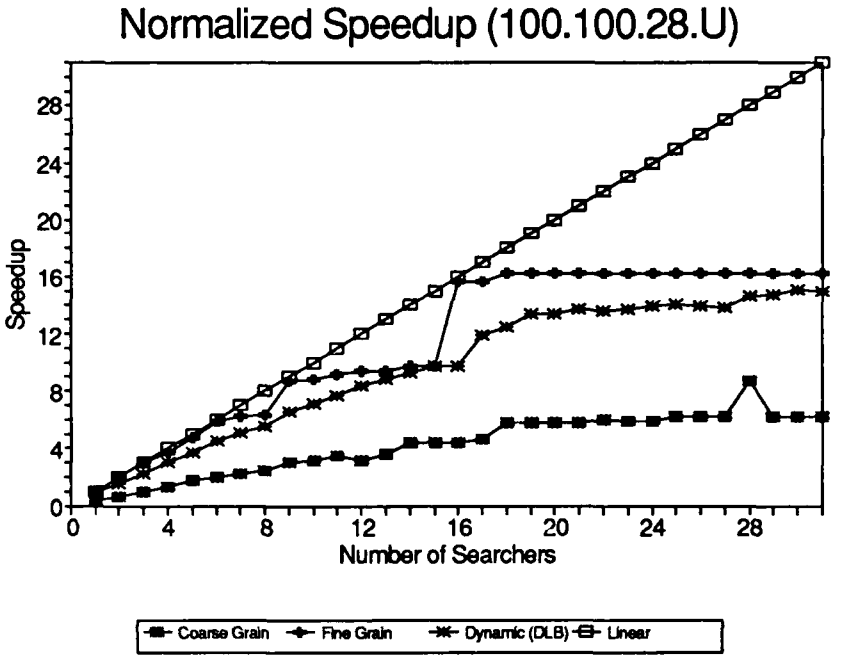



Figure 6.1. Normalized Speedup for Test Matrix 100.100.28.U

As previously noted, processor idle-time alone is not an accurate indicator of the performance of parallel algorithms. For example, an examination of Figures 6.1 to 6.5 clearly contradicts the previous conclusion that the DLB algorithm is better than the fine grain algorithm which is better than the coarse grain algorithm. Figure 6.1 shows that the fine grain algorithm is better than both the DLB and the coarse grain algorithms. On the other hand, Figure 6.4 shows that the coarse grain algorithm performs better than the other two algorithms for this particular test case.

The difference is easily explained. Recall from Chapter IV that the coarse grain and the fine grain versions of the SCP use different breadth-first expansion algorithms. Given that NP-complete problems are inhomogeneous, the different expansion algorithms produce radically different search graphs; hence, the difference in performance between the coarse grain and fine grain algorithms is unpredictable. In fact, one could argue that the

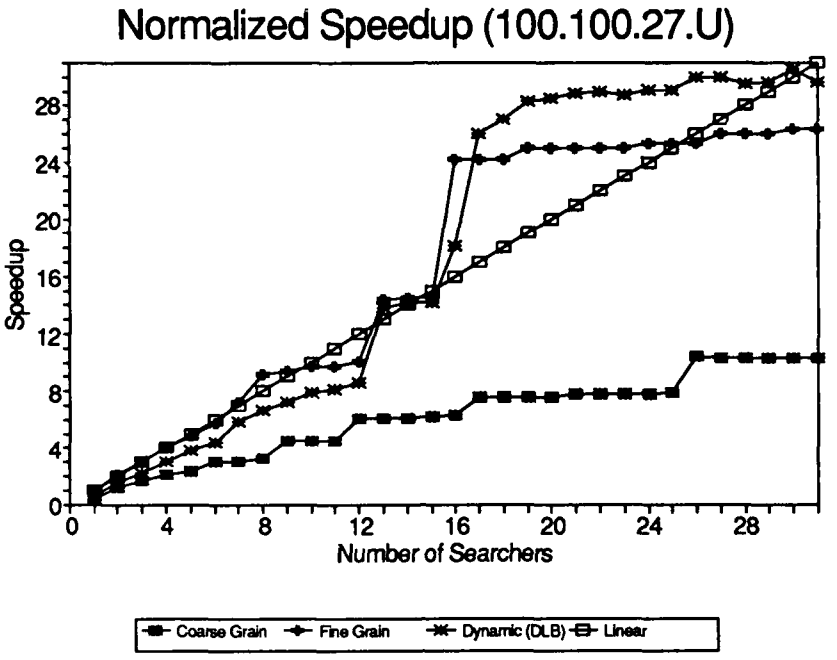



Figure 6.2. Normalized Speedup for Test Matrix 100.100.27.U

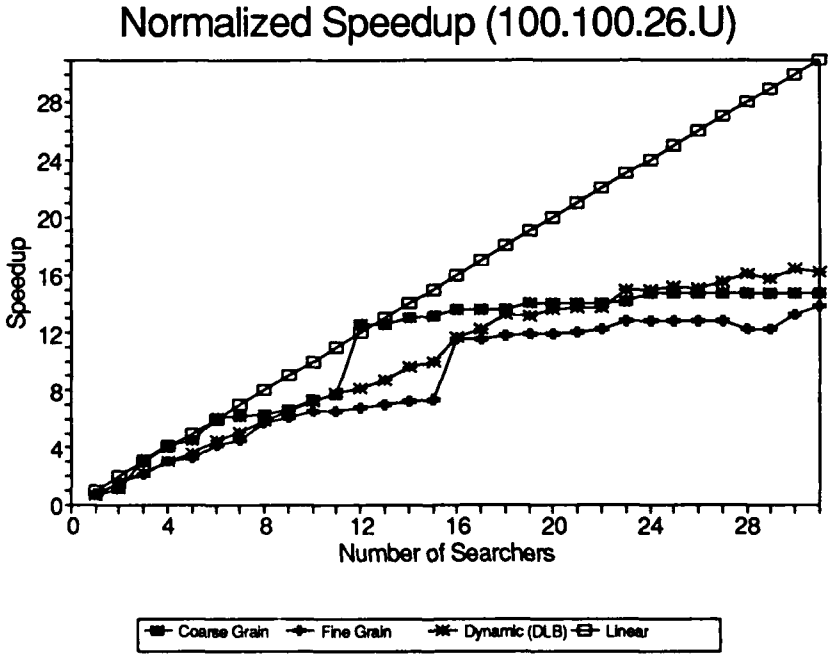



Figure 6.3. Normalized Speedup for Test Matrix 100.100.26.U

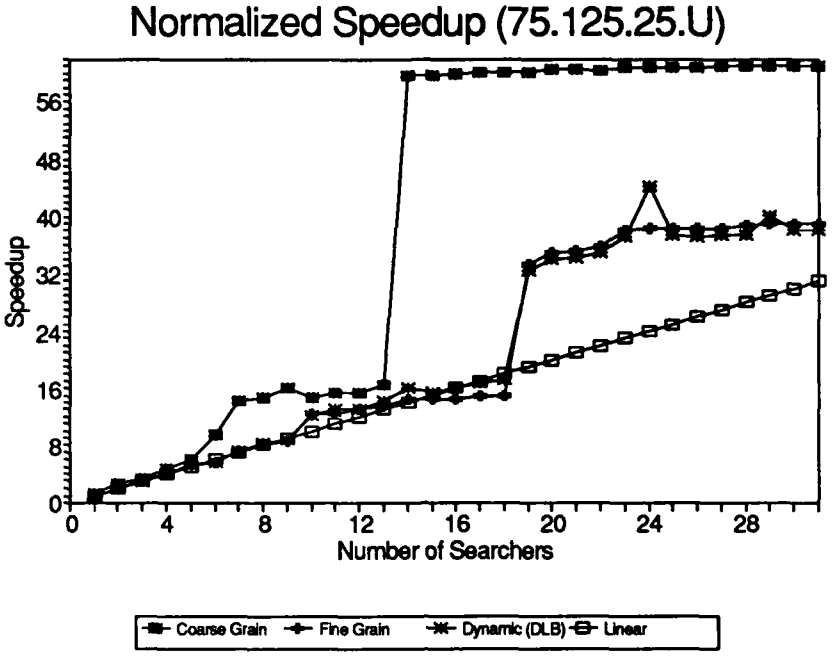



Figure 6.4. Normalized Speedup for Test Matrix 75.125.25.U

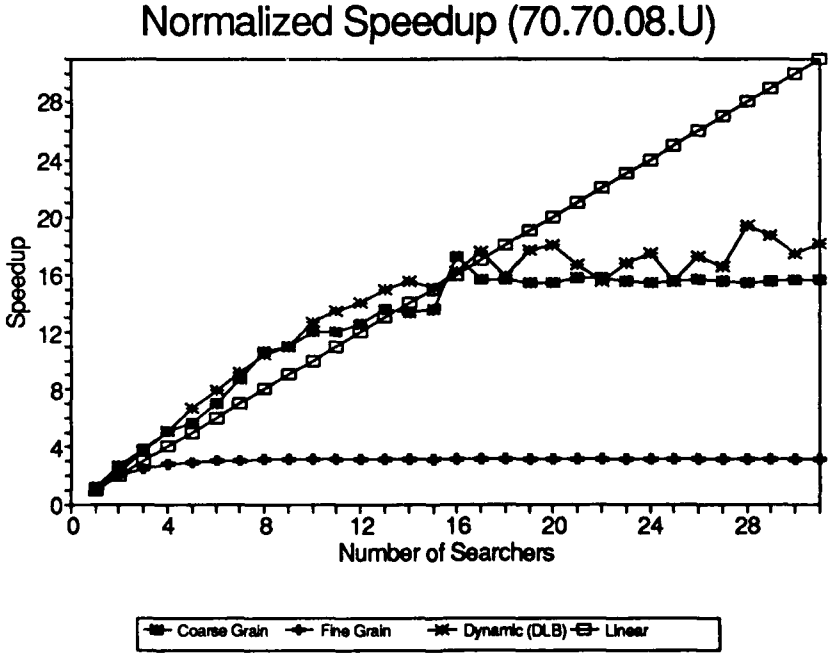



Figure 6.5. Normalized Speedup for Test Matrix 70.70.08.U

two algorithms are searching entirely different problems since the search graphs produced for the same problem may be completely different.

The expansion algorithm affects both the solution time and the idle-time; however, the processor idle-time is affected more by the allocation of the initial subgraphs than by the expansion algorithm. Since subgraphs in the coarse grain algorithm are statically allocated, processor idle-times are typically longer with the coarse grain algorithm than with the dynamically allocated fine grain algorithm. Even so, an examination of the raw data reveals that the coarse grain expansion algorithm usually results in a quicker best cover time (i.e., it finds the optimal cover before the fine grain expansion algorithm).

The DLB algorithm was developed to fu ther decrease the maximum processor idle-time and to improve the efficiency of the search algorithm. Notice in Figure 6.1 that the fine grain algorithm is consistently faster than the DLB algorithm. Either the DLB is inefficient or the fine grain algorithm is highly efficient for this problem instance. In this particular problem, the fine grain expansion algorithm balances the load from the beginning of the search. Any additional load balancing (e.g., dynamic load balancing) simply steals CPU cycles from the search algorithm and delays the completion of the search.

The DLB algorithm is not necessarily inefficient; however, it does include additional code to dynamically share portions of a processor's search graph. Even though the timing data obtained from the node processors indicates an extremely small percentage of time devoted to the dynamic load balancing process, the data does not indicate the total processor time devoted to the token process. This time is significant in some problem instances as shown in Figures 6.1, 6.2, 6.3, and 6.4. In each of these graphs, the speedup of the fine grain algorithm closely parallels the speedup of the DLB algorithm. Furthermore, notice that the fine grain algorithm is frequently more efficient than the DLB algorithm even though the performance data from the searching processors indicates the DLB algorithm did in fact share subgraphs between searching processors. Two reasons for the DLB's apparent inefficiency are: 1) the token process is stealing too much time from the search process, 2) the searching processors are spending too much time partitioning and sending subgraphs to other processors. Unfortunately, the `mclock()` function does not provide a method to compute the CPU time consumed by the separate token process; hence, another method

must be found to measure process time. The second reason suggests that the processors are partitioning the subgraph at too low a level in the search tree and a heuristic algorithm is required to prevent such low-level partitioning.

Despite the previous figures, Figure 6.5 shows that the DLB algorithm does work. For this specific problem, the fine grain expansion algorithm only creates 19 subgraphs due to limitations in the expansion algorithm. In effect, this is a coarse grain partitioning of the initial search graph. Since only 19 subgraphs are developed, the processors quickly become idle and the efficiency of the search suffers. With the DLB algorithm, the idle processors immediately receive a subgraph from the working processors and contribute to the search. Had the dynamic load balancing algorithm not been effective, the DLB's speedup curve would have paralleled the fine grain algorithm's curve as in previous graphs.

### *6.3 General Conclusions*

Given the preceding phenomena; namely,

- the maximum idle-time is not a true indicator of an algorithms' performance;
- in specific instances, the coarse grain algorithm is more efficient than the fine grain or DLB algorithms; and
- the DLB algorithm is not necessarily more efficient than the fine grain algorithm;

are the research results inconclusive? Certainly not!

One of the objectives of this research was to investigate methods to parallelize NP-complete problems. Three methods are presented and a speedup is obtained for each. In fact, a super-linear speedup is obtained for four of the five test matrices. The possibility of super-linear speedup in branch-and-bound search problems was predicted by Lai and Sahni (40) but it is unclear whether anyone had confirmed this phenomenon via the test results from an actual implementation. This is not to say that the algorithms presented here routinely produce a super-linear speedup. On the contrary, one could develop many test cases which would quickly disprove such a statement. However, the algorithms presented here show a tendency to go super-linear for input test cases that require a substantial amount of time to solve with a serial algorithm. More research is required to ascertain

whether specific problem characteristics can be a priori exploited to obtain predictable super-linear speedup.

The performance increases presented here are the result of a different approach than that documented in much of the published literature (44, 51, 1, 46, 23). The typical approach to parallelizing an NP-complete problem seems to center around the existence of a centrally maintained priority queue containing unsolved subbranches. The processors receive a subgraph, further partition the subgraph, and then transmit the newly partitioned subgraphs back to the centrally maintained queue. Such an approach is communications intensive as shown by Quinn (51). The approach presented here is to partition the search space first and distribute the subgraphs to the individual processors. As such, the communications overhead becomes insignificant and the problem becomes compute bound. This simple but elegant approach to the initial load balancing is only possible because of the preordering (i.e., the construction of the SCP table) accomplished before the search. The result is a simple and highly efficient initial distribution of the load for many problem instances. The possibility of a similar preordering in other NP-complete problems is left for future researchers.

To date, much of the research into parallel branch-and-bound algorithms has focused on the traveling salesman problem. The research presented here contains the first known parallel implementation of the SCP. Given the general application of the SCP to many different problems and the results published in this document, applications based on a parallel SCP (e.g., weapon to target assignment, optimal resource scheduling, VLSI expression simplification, and information retrieval) could achieve considerable performance increases. Furthermore, the methods presented here show that it is possible to realize a performance increase using control and data structures centered around something other than a centrally maintained priority queue.

The results further indicate that the performance of a parallel NP-complete search is highly dependent on the method chosen to distribute or balance the load between the processors. The initial distribution of subgraphs accomplished by the parallel SCP algorithms, in many of the test cases, is sufficient to ensure a 'good' load balancing. However, as Figure 6.5 indicates, a dynamic load balancing algorithm is necessary in those instances

where the initial distribution fails to obtain the desired load balance. The dynamic load balancing algorithm developed for this research is a much simpler algorithm than those presented by Felten (23) or Ma (44). The algorithm employs a separate process to pass a token between the processors and to coordinate all load balancing. The separate token process is designed such that termination is easily detected and, in the absence of any other load balancing scheme, the DLB algorithm may provide acceptable performance.

### *6.4 Problems Encountered During Research*

The previous section interprets the final results, but does not discuss the problems encountered in designing and implementing the programs. The purpose of this section is to present significant problems encountered and interesting observations.

The majority of the data shown in the tables of Appendix D is based on time. The iPSC/2's `mclock()` function provides the time for either the host or node processors. Care must be taken when using this function since it operates differently on the host than on the nodes. For instance, it is easy to obtain a false parallel execution time using the host's `mclock()` function since the user's host process is periodically swapped during the parallel run and its clock suspended (35). Hence, the host's execution time from the `mclock()` function may return a time which is significantly less than the actual time spent solving the problem. On the other hand, the node processor `mclock()` function does not return the process time. It is based on the time since the last reset of the nodes and is more of a 'wall-clock' time. Since each process reads the same clock, the process times can not be added to obtain the total processor time spent on a problem.

The iPSC/2 limits the size of the host-to-node messages to 256 Kbytes but does not limit the message size between the node processors[1] (37). Since the host-to-node message size is limited, two routines were developed to send and receive large blocks of data. The respective routines are called `SendVector()` and `ReceiveVector()` and are contained in file `msgio.c` located in the directory with the parallel SCP programs. These two routines

[1]The message size between nodes is limited only by the amount of node processor memory.

operate similar to `csend()` and `crecv()` except that they send/receive a maximum message size of 256 Kbytes.

The procedure for compiling node programs with floating-point arithmetic is somewhat buried in the iPSC/2 manuals. Any reference to a floating-point number requires the object files to be compiled and linked with the **-sx** switch if that program is to be executed on a node containing a scalar processor. The switch is not required if the program executes on the host processor. Little information is provided here concerning the compiling and linking of C programs for the iPSC/2; however, complete information is contained in the iPSC/2 manuals listed in the bibliography (33, 34, 35, 36, 37, 8).

### *6.5 Recommendations for Further Research*

As with any research project, the investigation never ends! The following list of recommendations is provided to further extend this research topic:

1. Develop a parallel algorithm to build the SCP table. The construction of the table is currently accomplished on the controller by a serial algorithm. The amount of time to build the table is less than 0.5 seconds for any of the test problems listed in Table D.1; hence, only insignificant improvement in the overall SCP solution time can be realized by the creation of a parallel SCP table construction algorithm. Even so, the design and implementation of such an algorithm may be of academic importance as it is a parallel matrix manipulation problem.
2. Add an option to save state information. The state information is easily obtained from the programs and may be of interest for graphic animation of the search tree.
3. Build the SCP table such that the columns are sorted on their cost as well as their cardinality. The current algorithms only sort on cost; hence, the columns for unit cost matrices are not sorted. It may be beneficial to sort the columns according to decreasing order on the column cardinality. Such an ordering would put the columns covering more rows at the beginning of the blocks in the table.
4. Investigate other methods of partitioning the initial input search graph. Perhaps use the coarse grain breadth-first expansion method for the fine grain and dynamic

load balanced algorithms. Since the best cover times in Table D.5 are typically less than the corresponding best cover times in either Tables D.6 or D.7, it is reasonable to assume that the coarse grain expansion algorithm is more likely to lead to a quicker solution. Suggest the coarse grain expansion algorithm be implemented with dynamic allocation (i.e., partition the subgraphs according to the coarse grain expansion algorithm and dynamically assign the subgraphs to the searching processors).

5. Determine why the dynamic load balancing algorithm occasionally performs worse than the fine grain algorithm. As previously stated, it may be that the token process is consuming too much CPU time or that the dynamic load balancing algorithm is dividing the search tree at too low a level.

6. Investigate the use of the dynamic load balancing algorithm as the sole method of distributing the load between the processors. Figure 6.5 seems to indicate that the dynamic load balancing algorithm may be sufficient to obtain an acceptable speedup in many problem instances.

7. Apply the parallel SCP algorithms or concepts to 'real-world' problems. Given the demonstrated performance of the SCP algorithms, it would seem reasonable to expect performance increases in many problems for which the SCP is particularly suited. Examples of such problems are presented in Appendix E.

8. Investigate the existence of specific problem characteristics which may lead to superlinear speedup.

## *6.6 Summary*

As stated on pages 2-23 and 2-25 of Chapter II, NP-complete problems are inhomogeneous; therefore, it is extremely difficult to make any a priori predictions concerning the nature of the search. In other words, the search may finish quickly or may require exponential time. A parallel implementation of an NP-complete problem can not change this fact.

This document has presented the design and implementation for a serial and three parallel SCP algorithms. The basic design philosophy has been to develop the serial algorithm first followed by a parallel decomposition. The SCP is an optimal search problem; hence, it is most naturally parallelized with a data parallelism approach. UNITY designs are developed and mapped to the iPSC/2 architecture per the design process outlined in Chandy and Misra (16).

A coarse grain algorithm with static allocation of the initial subgraphs is designed and implemented. Preliminary testing indicates that the algorithm is effective but that many of the searching processors are idle for extended periods of time. Therefore, in an attempt to decrease the processor idle-time, a fine grain algorithm with dynamic allocation is developed. The fine grain algorithm typically exhibits less processor idle-time but test results still indicate that valuable CPU time is being wasted due to idle searching processors.

Finally, a dynamic load balanced algorithm is designed and implemented. An inspection of individual processor idle-time indicates that this algorithm is more efficient than both the coarse grain and fine grain algorithms. However, as above analysis has shown, such a conclusion is invalid since given a 'good' initial division of the search graph by the expansion algorithm, the use of a dynamic load balancing algorithm is detrimental to the search. Even so, the dynamic load balancing algorithm was shown to improve the performance of a poorly balanced search.

# Appendix A. *Serial SCP Program Structure Charts and ADTs*

Chapter III stated that many of the routines for a parallel implementation of the SCP were actually serial; therefore, a serial version of the SCP was implemented as a base for the parallel implementation. The design is given in terms of UNITY programs and pseudo-code algorithms. This appendix contains additional documentation on the serial routines.

The first section contains structure charts for the general design and the second section contains abstract data types (ADTs) for the individual routines. The structure charts show the calling and called routines. A table of contents is included with a brief explanation of each chart. The ADTs show the data passing between the routines and, since these ADTs are not standard, the notation is explained at the beginning of the section. Also, a table of contents and brief explanation are given for each ADT.

## *A.1 Serial Set Covering Problem Structure Charts*

An explanation of the notation is required. Routine names of the form `Serial3cp` (i.e., routine names with no parenthesis) are not actual program names, but references to other figures. Routine names of the form `PrintHelp()` are actual routine names.

**Serial SCP Structure Charts:**

**Figure A.1** `Serial Scp` is the main controlling program and interface with the user. It calls routines to read the 0–1 matrix, implement the a priori reductions, build the initial data structures and table, search the matrix, and write the reordered/reduced matrix to a user specified file.

**Figure A.2** `Reductions` references all the serial reductions.

**Figure A.3** `QuickSort()` is a modified version of a quick sort routine obtained from Computer Innovations C86 compiler (20).

**Figure A.4** `Debug Routines` refers to routines which print the table data structure in various forms. This output is not normally available to the user. They are included for future work.

**Figure A.5** `SerialScpSearch()` is the actual search routine. It searches the input matrix using the table constructed by `BuildTable()` referenced in Figure A.1. `LowerBound()` is a lower bound test and `ExistDomSet()` is a dominance test.

**Figure A.6** `Stack Operations` refers to a semi-generic set of stack operations.

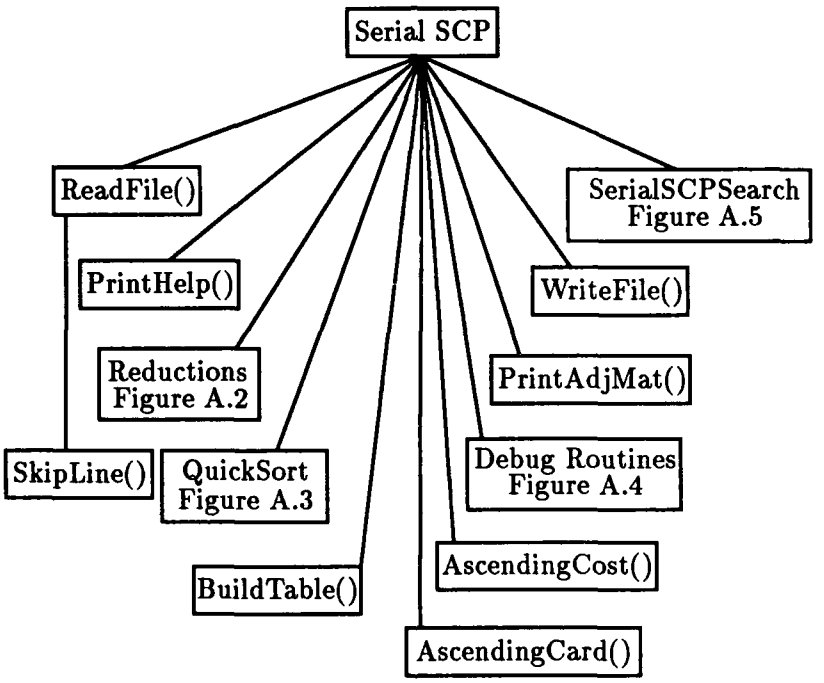



Figure A.1. Serial SCP Structure Chart

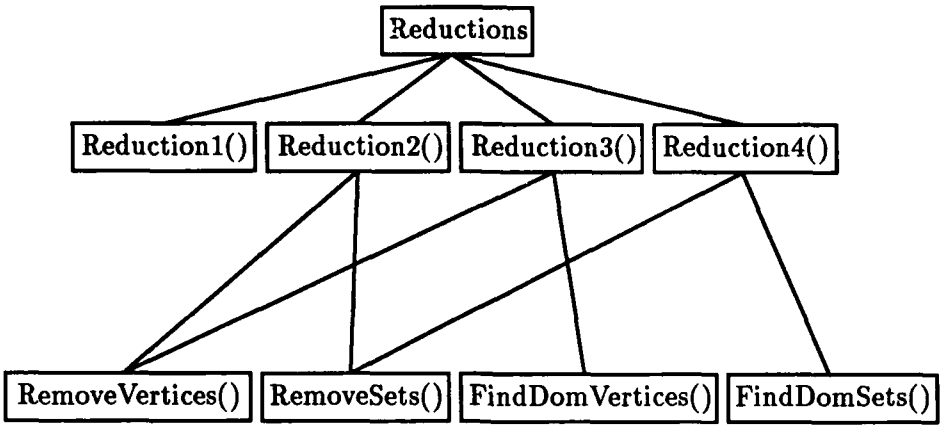



Figure A.2. Serial SCP Reductions Structure Chart

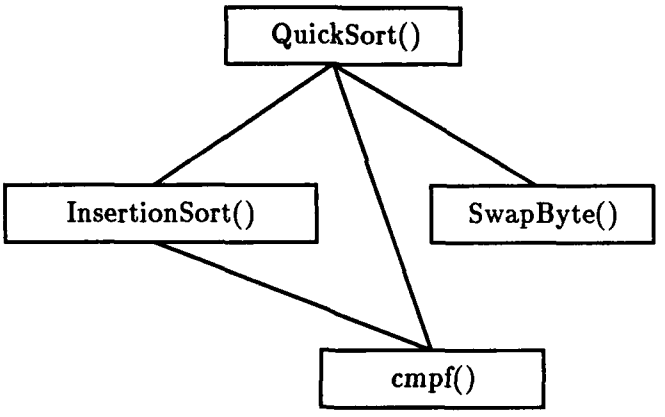



Figure A.3. Serial Sorts Structure Chart

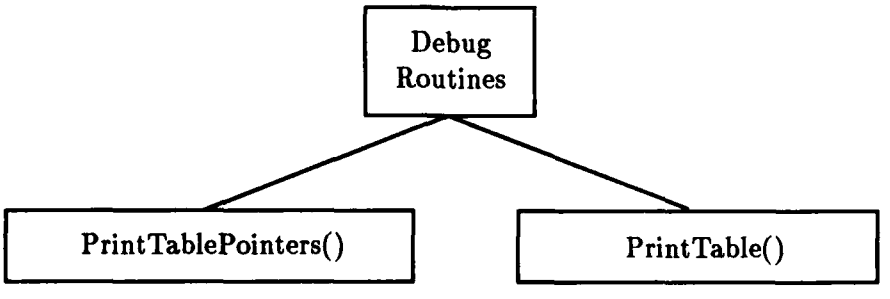



Figure A.4. Serial SCP Debug Routines Structure Chart

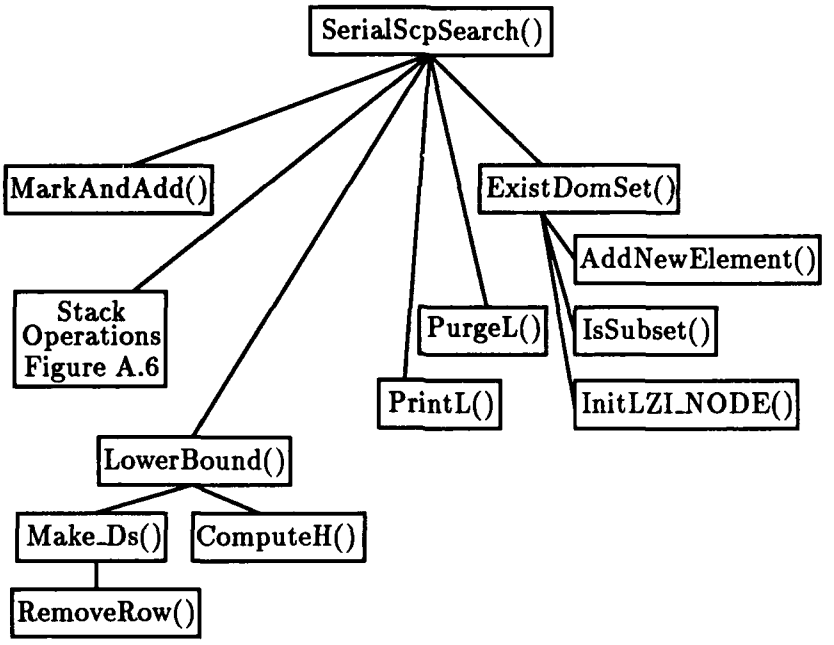



Figure A.5. Serial SCP Search Structure Chart

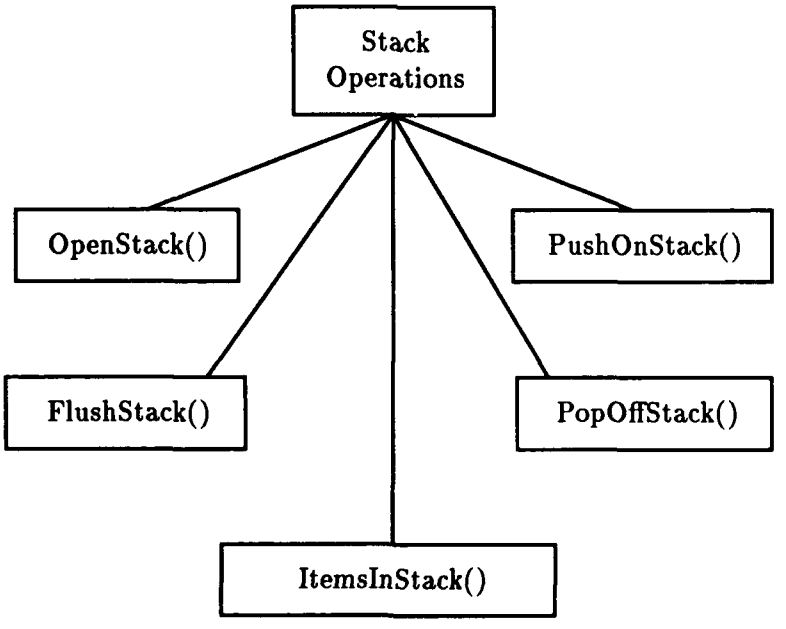



Figure A.6. Stack Structure Chart

## *A.2 Serial Set Covering Problem ADTs*

These ADTs are included to help future engineers modify or use this software. The ADT name (e.g., **CMPF**) is the name of the file containing the routines. The routines contained within a particular ADT show the flow of data into or out of the routine. Standard data types such as **int**, **boolean**, and **char** are used. In an attempt to make the data types more explicit, data types more descriptive than say, **record** or **array**, are used and the name of the variable is given. This structure is similar to the prototypes used in ANSI C. **TABLE_NODE** is an individual record of the table shown in Figure 3.3 on page 3-11. For example:

| | | | |
|---|---|---|---|
| 0 | 0 | Down | Right |

in Block 0 is a **TABLE_NODE**. **VERTEX** and **SET** refer to the vertex and set records shown in Figures 3.4 and 3.5 on page 3-12. An **S_NODE** is an entry on the stack. Figure 4.5 on page 4-9 shows the stack which is constructed as a singly-linked list where the items on the stack are pointers to a data structure.

**Serial SCP ADTs:**

**ADT A.1 CMPF** — Contains the comparison functions for the generic sorts. The sorts call these routines to compare the items passed to it for sorting. In this manner, the sort routines can sort any type of data.

**ADT A.2 DOMTEST** — The dominance test routines. See Section E.6.6 for an explanation of the dominance test.

**ADT A.3 FILE** — This ADT contains the file handling routines.

**ADT A.4 FSUBSET** — The reductions (ADT A.8) use these routines to find and mark rows or columns which are dominated by other rows and columns.

**ADT A.5 LBOUND** — The lower bound test routines. See Section E.6.7 for an explanation of the lower bound test.

**ADT A.6 PRINTADJM** — Prints the 0–1 matrix to a file.

**ADT A.7 PRTHELP** — Prints a help screen to the monitor.

**ADT A.8** `REDUCE` — Reduce the 0–1 input matrix.

**ADT A.9** `REMOVEVS` — Remove the rows or columns for the reduction routines.

**ADT A.10** `SCP` — The `main` routine. It coordinates all action.

**ADT A.11** `SRLSORTS` — Three generic, serial sort routines.

**ADT A.12** `STACK` — A semi-generic implementation of a stack.

**ADT A.13** `TABLE` — Build the table to assist in the search.

## ADT A.1

```
structure CMPF
      declare
            DescendingCard(VERTEX *Vertex1, VERTEX *Vertex2) → Int
            AscendingCost(SET *Set1, SET *Set2) → Int
end CMPF
```

## ADT A.2

```
structure DOMTEST
      declare
            ExistDomSet(LZI_NODE **L, int Zprime, TABLE_NODE *NextSet,
                        int CostOfNextSet, int Nverts, int Nsets, int *AdjMat,
                        VERTEX *Vertex, SET *Set) → BOOLEAN, L
            AddNewElement(LZI_NODE *Lptr, int *NewEprime, int NewCard) → Lptr
            IsSubset(int *E, EL_NODE *Element, int CardE) → BOOLEAN
            InitLZI_NODE(LZI_NODE *Node, int Z, EL_NODE *El, int NumEl
                         LZI_NODE *Nxt) → Node
            PurgeL(LZI_NODE *L) → L
            PrintL(LZI_NODE *L)
end DOMTEST
```

## ADT A.3

```
structure FILE
      declare
            BOOLEAN ReadFile(char *FileName, int Nverts, int Nsets, int *AdjMat,
                             VERTEX *Vertex, SET *Set) → BOOLEAN, Nverts,
                             Nsets, AdjMat, Vertex, Set
            SkipLine(FILE *Stream) → Stream
            BOOLEAN WriteFile(char *FileName, int Nverts, int Nsets, int *AdjMat,
                              VERTEX *Vertex, SET *Set) → BOOLEAN
end FILE
```

**ADT A.4**

```
structure FSUBSET
      declare
            FindDomVertices(int NumVertices, int Nsets, int *AdjMat, VERTEX *Vertex)
                        → Vertex
            FindDomSets(int NumSets, int Nverts, int Nsets, int *AdjMat,
                        VERTEX *Vertex, SET *Set) → Set
end FSUBSET
```

**ADT A.5**

```
structure LBOUND
      declare
            LowerBound(TABLE_NODE *Table, TABLE_NODE *TheNextSet, int Z,
                  int Zhat, int Nverts, VERTEX *Vertex) → BOOLEAN
            Make_Ds(TABLE_NODE *TheNextBlock, int *Ri, int NumUncoveredRows,
                  int **D, int **Dprime, int *NRows, int *NCols, int Nverts, int Nsets,
                  int *AdjMat, VERTEX *Vertex, SET *Set) →
                  D, Dprime, NRows, NCols
            ComputeH(int *D, int *Dprime, int NumRows, int NumCols, int Z, int Zhat) →
                  Int
            RemoveRow(int *Ri, int *NumUncoveredRows, int Row) →
                  Ri, NumUncoveredRows
end LBOUND
```

**ADT A.6**

```
structure PRINTADJM
      declare
            void PrintAdjMat(int Nverts, int Nsets, int *AdjMat, VERTEX *Vertex,
                        SET *Set)
end PRINTADJM
```

**ADT A.7**

```
structure PRTHELP
      declare
            PrintHelp()
end PRTHELP
```

## ADT A.8

structure REDUCE
    declare
        Reduction1(int Nverts, VERTEX *Vertex) → BOOLEAN
        Reduction2(int **RemovedSets, *CostOfRemovedSets, int Nverts, int Nsets,
                int *AdjMat, VERTEX *Vertex, SET *Set) → RemovedSets,
                CostOfRemovedSets, Nverts, Nsets, AdjMat, Vertex, Set
        Reduction3(int Nverts, int *AdjMat, VERTEX *Vertex) →
                Nverts, AdjMat, Vertex
        Reduction4(int Nsets, int *AdjMat, VERTEX *Vertex, SET *Set) →
                Nsets, AdjMat, Vertex, Set
end REDUCE

## ADT A.9

structure REMOVEVS
    declare
        RemoveVertices(int Nverts, int Nsets, int *AdjMat, VERTEX *Vertex) →
                Int, Nverts, AdjMat, Vertex
        RemoveSets(int Nverts, int Nsets, int *AdjMat, VERTEX *Vertex, SET *Set) →
                Int, Nsets, AdjMat, Vertex, Set
end REMOVEVS

## ADT A.10

structure SCP
    declare
        main(int argc, char *argv[], int Nverts, int Nsets, int *AdjMat,
            VERTEX *Vertex, SET *Set) → Nverts, Nsets, AdjMat, Vertex, Set
        SerialScpSearch(TABLE_NODE, *Table, int *CostOfCoveringSets,
                int **CoveringSets, int Nverts, int Nsets, int *AdjMat,
                VERTEX *Vertex, SET *Set) →
                CostOfCoveringSets, CoveringSets, Vertex, Set
        MarkAndAdd(TABLE_NODE *TheNextSet, int *NumVerticesCovered,
                int **Cover, int CoverCost, int NextElement, int Nverts,
                int Nsets, int *AdjMat, VERTEX *Vertex, SET *Set) →
                Int, NumVerticesCovered, Cover, NextElement, Vertex, Set
end SCP

## ADT A.11

```
structure SRLSORTS
      declare
            MergeSort(char *Base_i, char *Base_j, int NumElements_i, int NumElements_j.
                  int ElementWidth, int (*cmpf)()) → Base_i, Base_j
            QuickSort(char *Base, int NumElements, int ElementWidth,
                  int (*cmpf)()) → Base
            InsertionSort(char *Base, int NumElements, int ElementWidth,
                  int (*cmpf)()) → Base
            SwapByte(char *aptr, char *bprt, int count)
end SRLSORTS
```

## ADT A.12

```
structure STACK
      declare
            OpenStack(S_NODE **TopPtr) → TopPtr
            FlushStack(S_NODE **TopPtr) → TopPtr
            PushOnStack(S_NODE **TopPtr, ITEM *NewItemPtr) → TopPtr
            PopOffStack(S_NODE **TopPtr) → ITEM, TopPtr
            ItemsInStack(S_NODE *TopPtr) → Int
end STACK
```

## ADT A.13

```
structure TABLE
      declare
            BuildTable(TABLE_NODE **Table, int Nverts, int Nsets, int *AdjMat,
                  VERTEX *Vertex, SET *Set) → BOOLEAN, Table
            PrintTable(TABLE_NODE *Table, int Nverts, int Nsets, int *AdjMat,
                  VERTEX *Vertex, SET *Set)
            PrintTablePointers(TABLE_NODE *Table)
end TABLE
```

## Appendix B. *Parallel SCP Program Structure Charts and ADTs*

This appendix, similar to Appendix A, contains the structure charts and ADTs for the parallel software. Since a large part of the parallel routines are simply serial routines, several of the structure charts in this appendix may refer to structure charts in the Appendix A.

The first section contains the structure charts with the ADTs in the second section. Both sections contain a table of contents and brief description similar to Appendix A; therefore, an explanation of the notation is not repeated here except when different from the previous appendix.

### *B.1 Parallel Set Covering Problem Structure Charts*

**Parallel SCP Structure Charts:**

**Figure B.1** `Communications Chart` shows the communications that occur over the cube network. These programs are all `main` programs running on either different processors or as different processes. They only communicate via cube **send** and **recv** commands.

**Figure B.2** **scp** is the main program similar to the serial version of the SCP. In the parallel case, **scp** executes on the host processor and is the interface with the user and the file system. It reads the input data and sends it to the cube control process: **scpcntxx**. Data arriving from the control process is displayed to the console or written to a file.

**Figure B.3** **scpcntcg** is the control program for the coarse grain version of the parallel programs. It sends the input data to the searching processors and then waits for the optimal answer to return. When all searchers are finished, it sends the optimal solution and performance statistics to **scp**.

**Figure B.4** **scpndcg** is the coarse grain controlling routine for the coarse grain version of the searcher program. It builds a table and a list of subgraphs from

the input matrix. Each searcher then executes its search on its predetermined subgraphs. Upon completion, it sends performance statistics to **scpcntcg**.

**Figure B.5** **sscpnode** is the controlling routine for the serial search program executing on a cube processor. This program is basically the same as Figure A.1 in Appendix A without the user interface and file I/O routines. It is included so that a better comparison between the serial and parallel program times can be obtained.

**Figure B.6** **Parallel Sorts** is a generic version of a parallel bitonic merge sort. The parallel sort is controlled by the controller and each searcher participates as directed.

**Figure B.7** **Reductions** are parallel versions of the a priori reductions.

**Figure B.8** **ScpSearch()** is the parallel search called by **scpndcg**. It is a modified version of **SScpSearch** which includes logic for the global best cost and communications with the controller.

**Figure B.9** **Queue Operations** contains a semi-generic set of queue operations.

**Figure B.10** **SScpSearch()** is the actual serial search called by **sscpnode**. It executes the same search as **SerialScpSearch()** in Figure A.5 of Appendix A.

**Figure B.11** **scpcntfg** is the controller for the fine grain parallel search. It builds subgraphs and ships them out to the searchers upon request. When all subgraphs are allocated, it basically waits for all searchers to finish and then sends the performance statistics and optimal solution to **scp**.

**Figure B.12** **scpndfg** is the fine grain controlling routine for the **ScpSearch()** of Figure B.8. It requests subgraphs from the controller and then passes the subgraph to **ScpSearch()**. When it no longer receives subgraphs, **scpndfg** sends its performance statistics to **scpcntfg**.

**Figure B.13** **scpndlb** is the dynamic load balanced parallel version of the SCP. It is slightly different than **scpndfg** in that it calls a different search routine.

**Figure B.14** **ScpSearch()** is the dynamic load balanced searcher program. It is different than the **ScpSearch()** in Figure B.8 because it must call a load bal-

ancing routine to transfer subgraphs to other searching processors. Other than the load balance routine, it is the same.

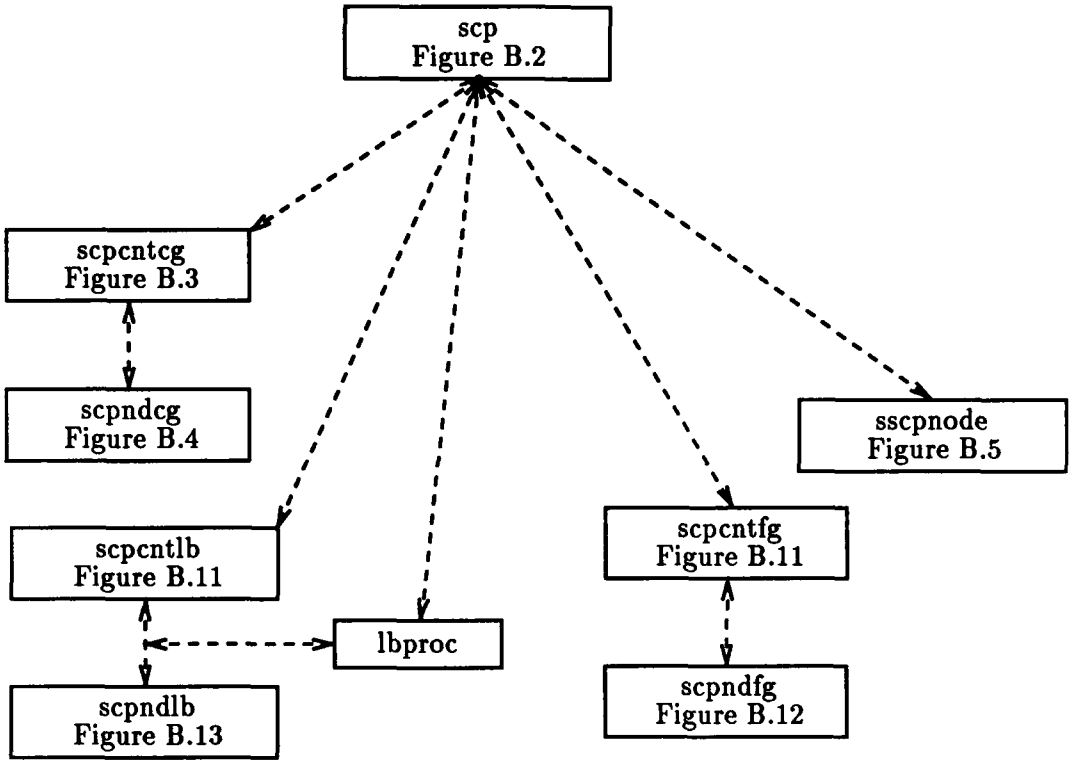



Figure B.1. Parallel SCP Communications Structure Chart

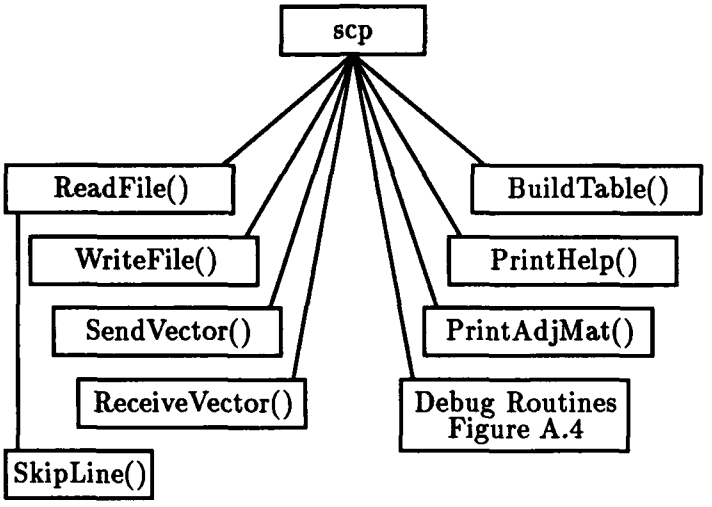



Figure B.2. Parallel SCP Host Structure Chart

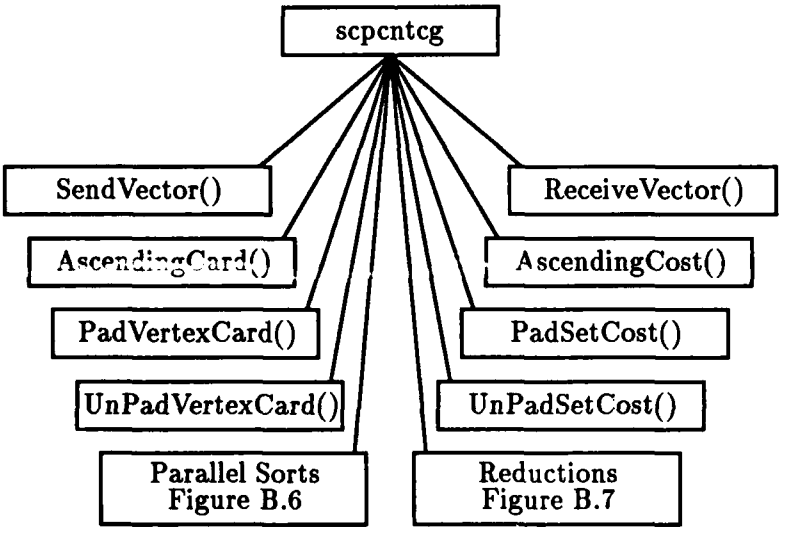



Figure B.3. Parallel Coarse Grain SCP Control Node Structure Chart

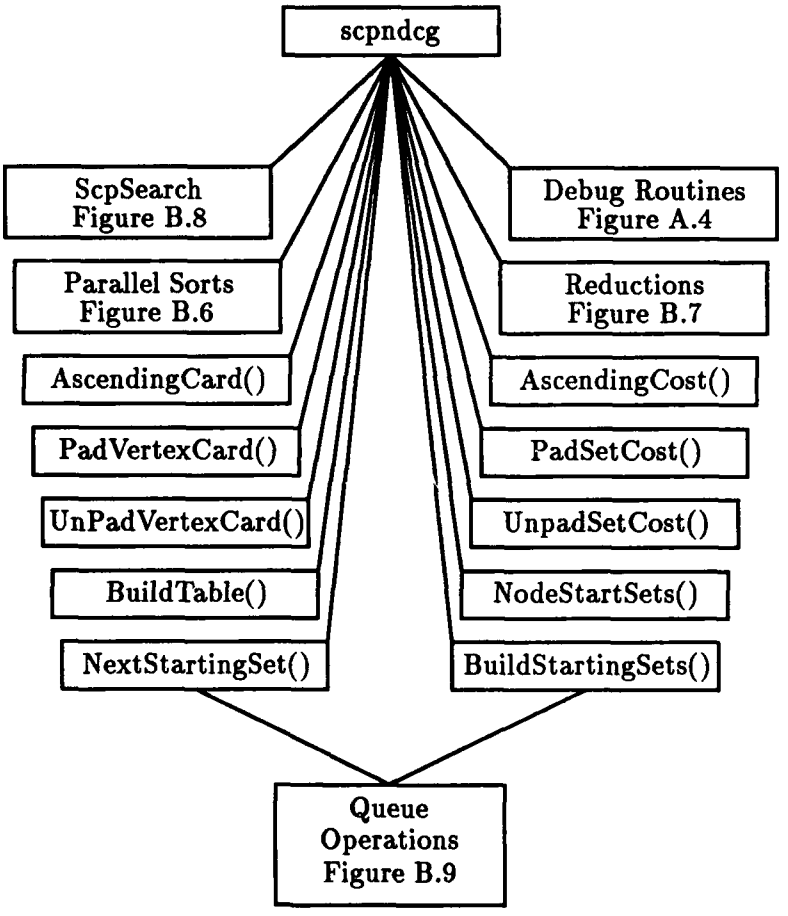



Figure B.4. Parallel Coarse Grain SCP Searcher Node Structure Chart

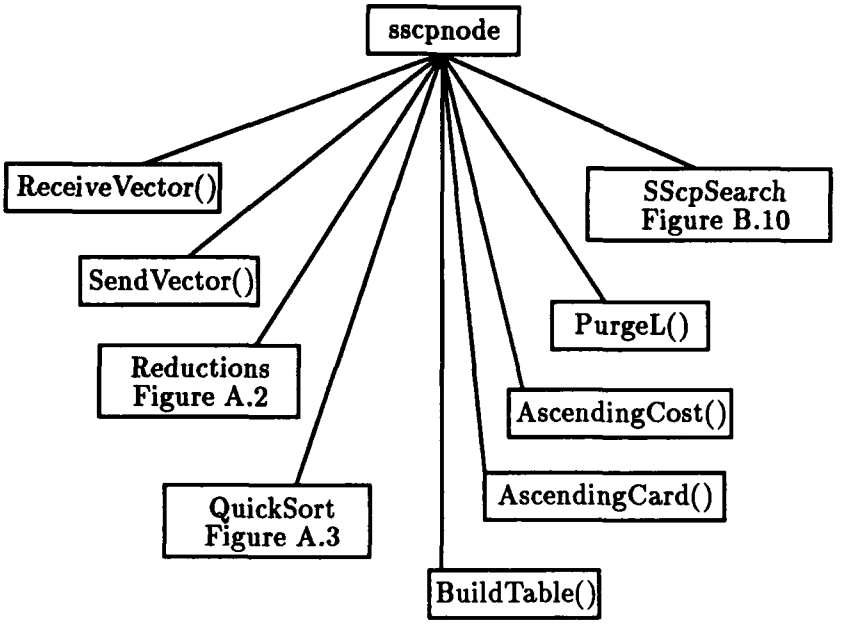



Figure B.5. Node Processor Serial SCP Structure Chart

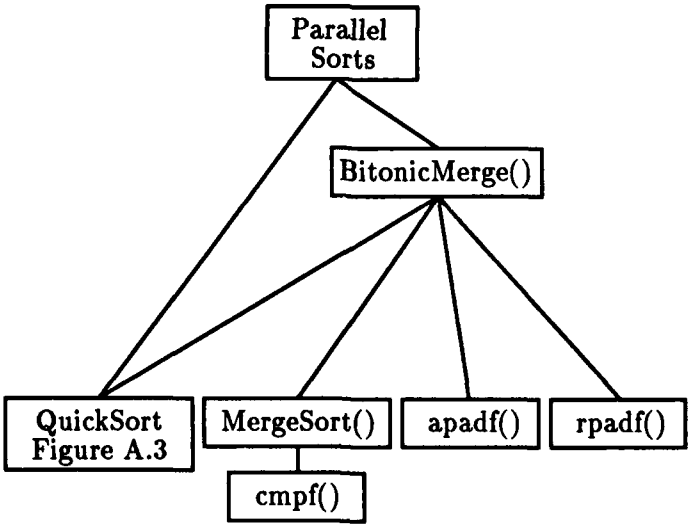



Figure B.6. Parallel Sorts Structure Chart

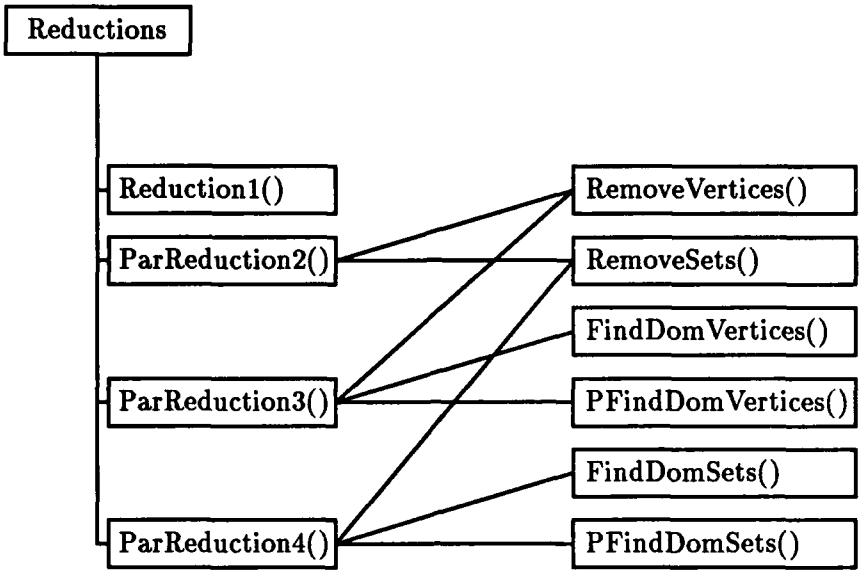



Figure B.7. Parallel SCP Reductions Structure Chart

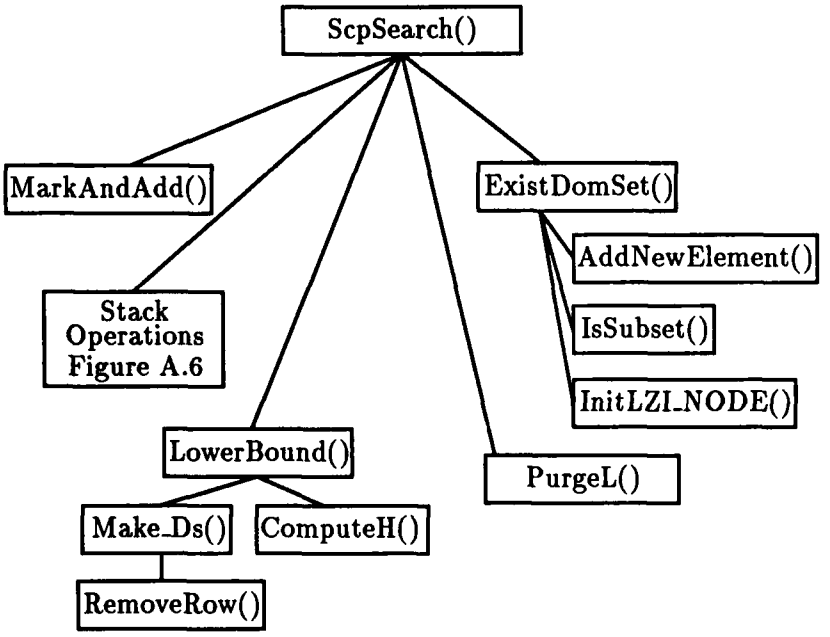



Figure B.8. Parallel Coarse Grain SCP Search Structure Chart

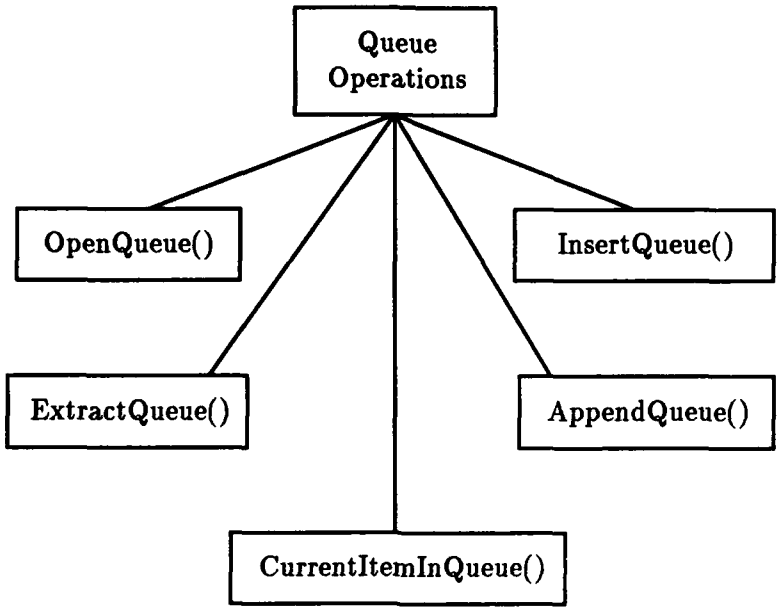



Figure B.9. Queue Structure Chart

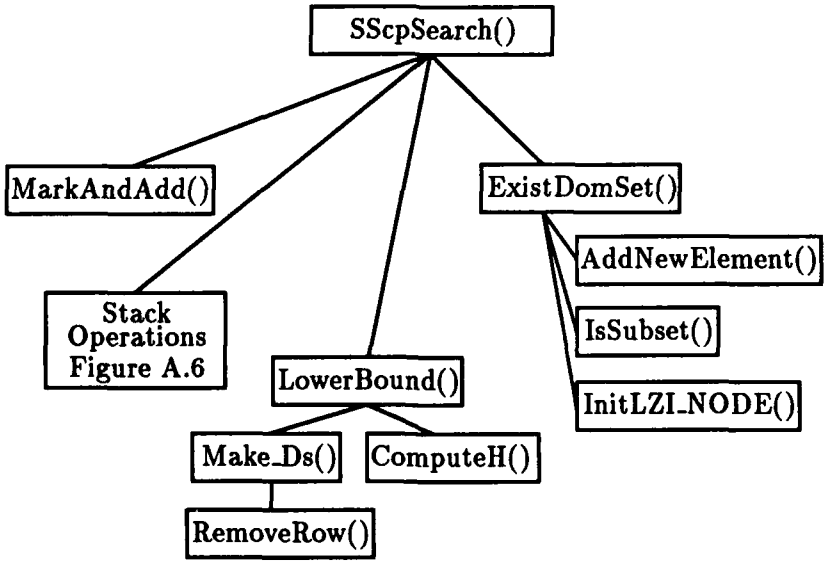



Figure B.10. Node Processor Serial SCP Search Structure Chart

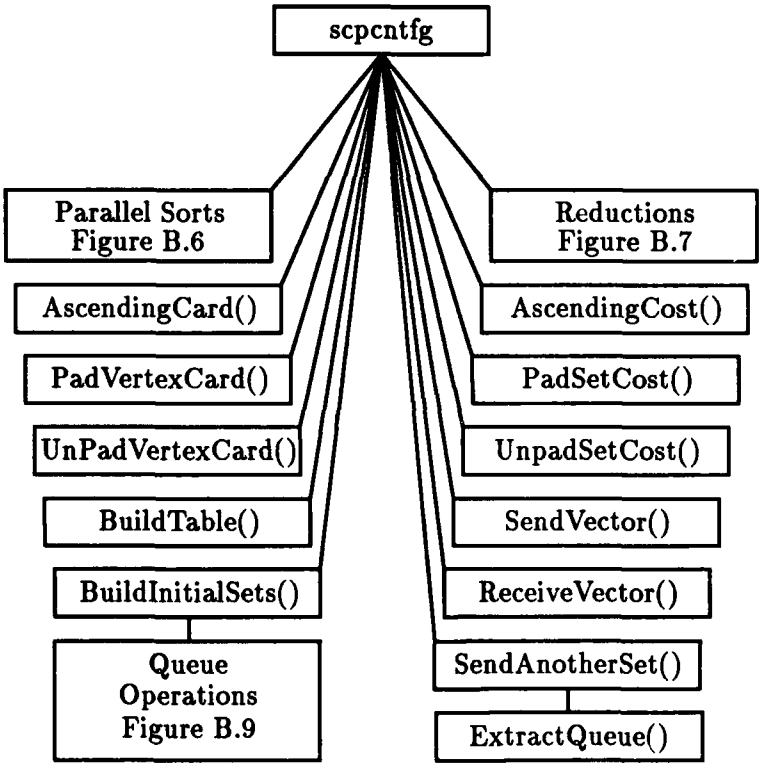



Figure B.11. Parallel Fine Grain SCP Control Structure Chart

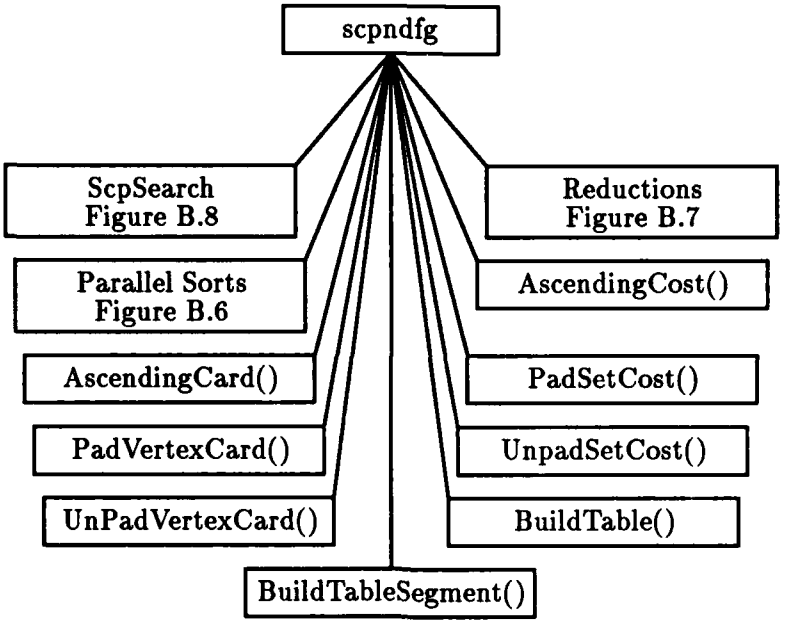



Figure B.12. Parallel Fine Grain SCP Searcher Node Structure Chart

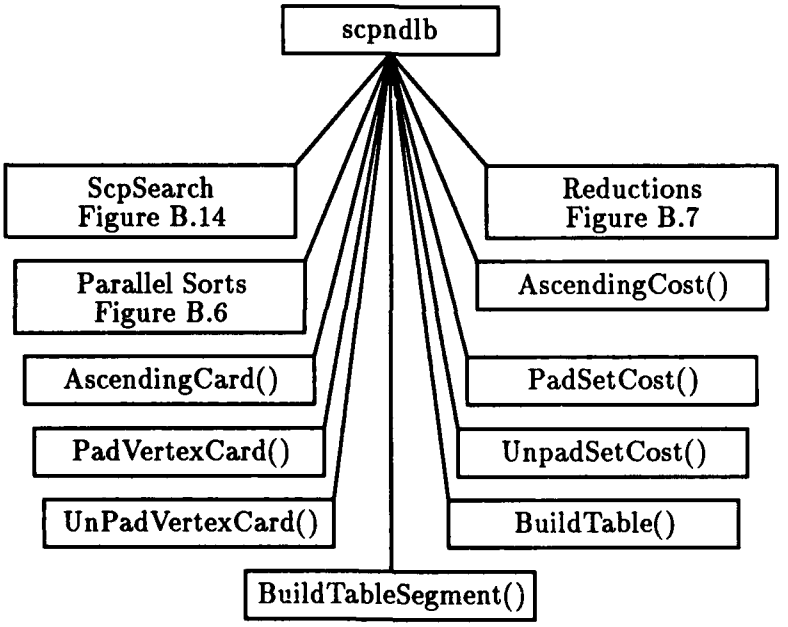



Figure B.13. Dynamic Load Balancing SCP Searcher Node Structure Chart

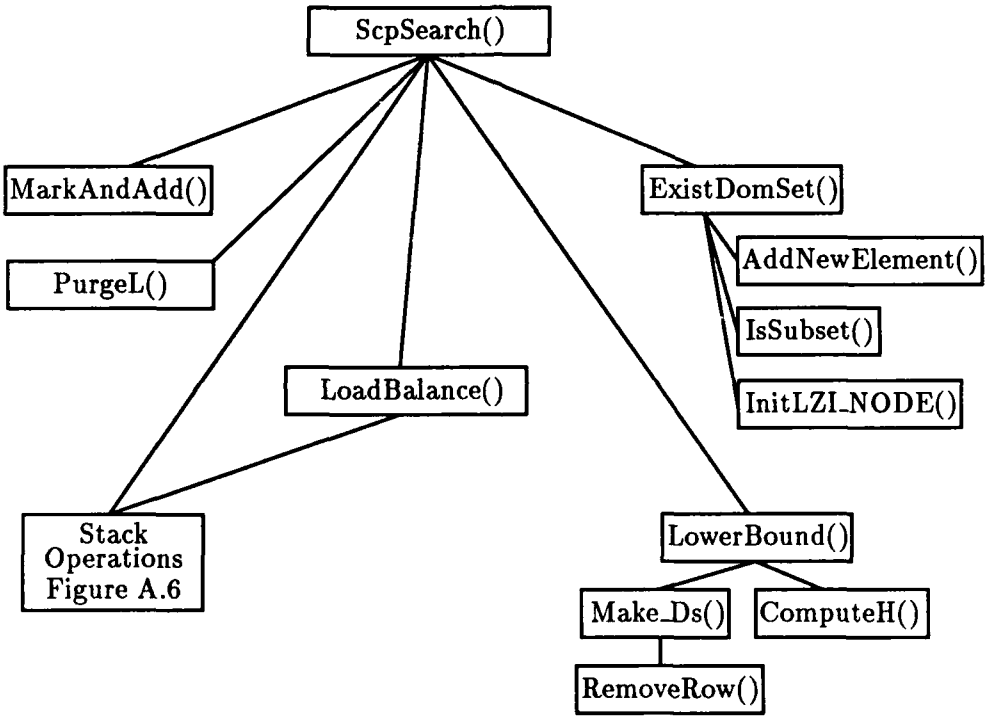



Figure B.14. Dynamic Load Balancing SCP Search Structure Chart

## *B.2 Parallel Set Covering Problem ADTs*

The ADTs are similar to those in Appendix A; however, a new data type has been added. The `Q_NODE` data type is a node in the queue. It has basically the same structure as `S_NODE` given in Appendix A. The major difference is that for the queue a pointer to the front and rear is required so that an item can be appended to the rear of the queue and removed from the front of the queue. Figure 4.7 on page 4-13 illustrates the queue data structure.

### Parallel SCP ADTs:

**ADT B.1** `CMPF` — Contains the comparison functions for the generic sorts. The sorts call these routines to compare the items passed to it for sorting. In this manner, the sort routines can sort any type of data.

**ADT B.2** `DOMTEST` — The dominance test routines. See Section E.6.6 for an explanation of the dominance test.

**ADT B.3** `FILE` —This ADT contains the file handling routines.

**ADT B.4** `FSUBSET` — The serial reductions (ADT B.15) use these routines to find and mark rows or columns which are dominated by other rows and columns.

**ADT B.5** `FSUBSETS` — The parallel reductions (ADT B.11) uses these routines to find and mark rows or columns which are dominated by other rows and columns. It contains modified versions of the routines in ADT B.15.

**ADT B.6** `LBOUND` — The lower bound test routines. See Section E.6.7 for an explanation of the lower bound test.

**ADT B.7** `LBPROC` — The dynamic load balancing routine. It is a separate process; hence, the ADT.

**ADT B.8** `MARKNADD` — Mark the rows as covered, the sets as used, and add the set to the list of covering sets.

**ADT B.9** `MSGIO` — Send and receive large, contiguous blocks of data. Needed between the node processors and the host processor.

**ADT B.10** `PARSORTS` — Parallel sort routines.

**ADT B.11** `PREDUCE` — Parallel a priori reductions.

**ADT B.12** `PRINTADJM` — Prints the 0–1 matrix to a file.

**ADT B.13** `PRTHELP` — Prints a help screen to the monitor.

**ADT B.14** `QUEUE` — A semi-generic implementation of a queue.

**ADT B.15** `REDUCE` — Reduce the 0–1 input matrix.

**ADT B.16** `REMOVEVS` — Remove the rows or columns for the serial and parallel reduction routines.

**ADT B.17** `SCP` — The `main` routine. It coordinates all action.

**ADT B.18** `SCPCNTCG`— Coarse grain parallel controller routine.

**ADT B.19** `SCPCNTLB` — Dynamic load balanced parallel controller routine.

**ADT B.20** `SCPCNTFG` — Fine grain parallel controller routine.

**ADT B.21** *`SCPNDCG` — The coarse grain parallel node routine* (i.e., a searcher).

**ADT B.22** `SCPNDLB` — The dynamic load balanced parallel node routines.

**ADT B.23** `SCPNDFG` — The fine grain parallel node routines.

**ADT B.24** `SRLSORTS` — Three generic, serial sort routines.

**ADT B.25** `SSCPNODE` — Serial SCP search executed on a node processor.

**ADT B.26** `STACK` — A semi-generic implementation of a stack.

**ADT B.27** `STARTSET` — Builds the subgraphs.

**ADT B.28** `TABLE` — Build the table to assist in the search.

## ADT B.1

```
structure CMPF
      declare
            AscendingCard(VERTEX *Vertex1, VERTEX *Vertex2) → Int
            PadVertexCard(char *Base, int NumVertices) → Base
            UnpadVertexCard(char *Base, int NumVertices) → Int, Base
            AscendingCost(SET *Set1, SET *Set2) → Int
            PadSetCost(char *Base, int NumSets) → Base
            UnpadSetCost(char *Base, int NumSets) → Int, Base

end CMPF
```

## ADT B.2

```
structure DOMTEST
      declare
            ExistDomSet(LZI_NODE **L, int Zprime, TABLE_NODE *NextSet,
                        int CostOfNextSet, int Nverts, int Nsets, int *AdjMat,
                        VERTEX *Vertex, SET *Set) → BOOLEAN, L
            AddNewElement(LZI_NODE *Lptr, int *NewEprime, int NewCard) → Lptr
            IsSubset(int *E, EL_NODE *Element, int CardE) → BOOLEAN
            InitLZI_NODE(LZI_NODE *Node, int Z, EL_NODE *El, int NumEl
                         LZI_NODE *Nxt) → Node
            PurgeL(LZI_NODE *L) → L
            PrintL(LZI_NODE *L)
end DOMTEST
```

## ADT B.3

```
structure FILE
      declare
            BOOLEAN ReadFile(char *FileName, int Nverts, int Nsets, int *AdjMat,
                             VERTEX *Vertex, SET *Set) → BOOLEAN, Nverts,
                             Nsets, AdjMat, Vertex, Set
            SkipLine(FILE *Stream) → Stream
            BOOLEAN WriteFile(char *FileName, int Nverts, int Nsets, int *AdjMat,
                              VERTEX *Vertex, SET *Set) → BOOLEAN
end FILE
```

## ADT B.4

```
structure FSUBSET
      declare
            FindDomVertices(int NumVertices, int Nsets, int *AdjMat,
                            VERTEX *Vertex) → Vertex
            FindDomSets(int NumSets, int Nverts, int Nsets, int *AdjMat,
                        VERTEX *Vertex, SET *Set) → Set
end FSUBSET
```

## ADT B.5

```
structure FSUBSETS
      declare
            FindDomVertices(int NumVertices, int Nsets, int *AdjMat,
                              VERTEX *Vertex) → Vertex
            FindDomSets(int NumSets, int Nverts, int Nsets, int *AdjMat,
                        VERTEX *Vertex, SET *Set) → Set
            PFindDomVertices(int NumVertices, int NumTopVerts, int Nsets,
                              int *AdjMat, VERTEX *Vertex) → Vertex
            PFindDomSets(int NumSets, int NumTopSets, int Nverts, int Nsets,
                        int *AdjMat, VERTEX *Vertex, SET *Set) → Set
end FSUBSETS
```

## ADT B.6

```
structure LBOUND
      declare
            LowerBound(TABLE_NODE *Table, TABLE_NODE *TheNextSet, int Z,
                        int Zhat, int Nverts, VERTEX *Vertex) → BOOLEAN
            Make_Ds(TABLE_NODE *TheNextBlock, int *Ri, int NumUncoveredRows,
                        int **D, int **Dprime, int *NRows, int *NCols, int Nverts, int Nsets,
                        int *AdjMat, VERTEX *Vertex, SET *Set) →
                        D, Dprime, NRows, NCols
            ComputeH(int *D, int *Dprime, int NumRows, int NumCols, int Z,
                        int Zhat) → Int
            RemoveRow(int *Ri, int *NumUncoveredRows, int Row) →
                        Ri, NumUncoveredRows
end LBOUND
```

## ADT B.7

```
structure LBPROC
      declare
            main()
end LBPROC
```

## ADT B.8

```
structure MARKNADD
      declare
            MarkAndAdd(TABLE_NODE *TheNextSet, int *NumVerticesCovered,
                        int **Cover, int CoverCost, int *NextElement, int Nverts,
                        int Nsets, int *AdjMat, VERTEX *Vertex, SET *Set) →
                        Int, NumVerticesCovered, Cover, NextElement, Vertex, Set
end MARKNADD
```

### ADT B.9

```
structure MSGIO
      declare
            SendVector(int SendType, char *Vector, int NumElements, int ElementSize,
                  int NodeID, int ProcessID)
            ReceiveVector(int ReceiveType, char *Vector, int NumElements,
                  int ElementSize) → Int, Vector
end MSGIO
```

### ADT B.10

```
structure PARSORTS
      declare
            ParSort(int Order, char *Base, int NumElements, int ElementWidth,
                  int (*cmpf)(), int (*apadf)(), int (*rpadf)(), int CubeDim) →
                  Int, Base
            BitonicMerge(int Order, char *aArray, int NumElements, int ElementWidth,
                  int CubeDim, int (*cmpf)()) → aArray
end PARSORTS
```

### ADT B.11

```
structure PREDUCE
      declare
            Reduction1(int Nverts, VERTEX *Vertex) → BOOLEAN
            ParReduction2(int **RemovedSets, int *CostOfRemovedSets, int Nverts,
                  int Nsets, int *AdjMat, VERTEX *Vertex, SET *Set) →
                  Int, RemovedSets, CostOfRemovedSets,
                  Nverts, Nsets, AdjMat, Vertex, Set
            ParReduction3(int CubeDim, int Nverts, int Nsets, int *AdjMat,
                  VERTEX *Vertex) → Nverts, AdjMat, Vertex
            ParReduction4(int CubeDim, int Nverts, int Nsets, int *AdjMat,
                  VERTEX *Vertex, SET *Set) → Nsets, AdjMat, Vertex, Set
end PREDUCE
```

### ADT B.12

```
structure PRINTADJM
      declare
            void PrintAdjMat(int Nverts, int Nsets, int *AdjMat, VERTEX *Vertex,
                  SET *Set)
end PRINTADJM
```

### ADT B.13

```
structure PRTHELP
      declare
            PrintHelp()
end PRTHELP
```

**ADT B.14**

structure QUEUE
    declare
        OpenQueue(Q_NODE **FrontPtr, Q_NODE **RearPtr) → FrontPtr, RearPtr
        InsertQueue(Q_NODE **FrontPtr, Q_NODE **RearPtr,
                ITEM *NewItemPtr) → FrontPtr, RearPtr
        ExtractQueue(Q_NODE **FrontPtr, Q_NODE **RearPtr) →
                ITEM, FrontPtr, RearPtr
        CurrentItemInQueue(Q_NODE **FrontPtr, Q_NODE **NextNodePtr) →
                ITEM, FrontPtr, NextNodePtr
        AppendQueue(Q_NODE **ToFrontPtr, Q_NODE **ToRearPtr,
                Q_NODE **FromFrontPtr, Q_NODE ** FromRearPtr) →
                ToFrontPtr, ToRearPtr, FromFrontPtr, FromRearPtr
end QUEUE

**ADT B.15**

structure REDUCE
    declare
        Reduction1(int Nverts, VERTEX *Vertex) → BOOLEAN
        Reduction2(int **RemovedSets, *CostOfRemovedSets, int Nverts, int Nsets,
                int *AdjMat, VERTEX *Vertex, SET *Set) → RemovedSets,
                CostOfRemovedSets, Nverts, Nsets, AdjMat, Vertex, Set
        Reduction3(int Nverts, int *AdjMat, VERTEX *Vertex) →
                Nverts, AdjMat, Vertex
        Reduction4(int Nsets, int *AdjMat, VERTEX *Vertex, SET *Set) →
                Nsets, AdjMat, Vertex, Set
end REDUCE

**ADT B.16**

structure REMOVEVS
    declare
        RemoveVertices(int Nverts, int Nsets, int *AdjMat, VERTEX *Vertex) →
                Int, Nverts, AdjMat, Vertex
        RemoveSets(int Nverts, int Nsets, int *AdjMat, VERTEX *Vertex,
                SET *Set) → Int, Nsets, AdjMat, Vertex, Set
end REMOVEVS

**ADT B.17**

structure SCP
    declare
        main(int argc, char *argv[], int Nverts, int Nsets, int *AdjMat,
            VERTEX *Vertex, SET *Set) → Nverts, Nsets, AdjMat, Vertex, Set
end SCP

**ADT B.18**

```
structure SCPCNTCG
      declare
            main(int argc, char *argv[], int Nverts, int Nsets, int *AdjMat,
                  VERTEX *Vertex, SET *Set) → Nverts, Nsets, AdjMat, Vertex, Set
end SCPCNTCG
```

**ADT B.19**

```
structure SCPCNTLB
      declare
            main(int argc, char *argv[], int Nverts, int Nsets, int *AdjMat,
                  VERTEX *Vertex, SET *Set) → Nverts, Nsets, AdjMat, Vertex, Set
end SCPCNTLB
```

**ADT B.20**

```
structure SCPCNTFG
      declare
            main(int argc, char *argv[], int Nverts, int Nsets, int *AdjMat,
                  VERTEX *Vertex, SET *Set) → Nverts, Nsets, AdjMat, Vertex, Set
end SCPCNTFG
```

**ADT B.21**

```
structure SCPNDCG
      declare
            main(int argc, char *argv[], int Nverts, int Nsets, int *AdjMat,
                  VERTEX *Vertex, SET *Set) → Nverts, Nsets, AdjMat, Vertex, Set
            ScpSearch(TABLE_NODE *SetsToSearch, int *NodeBestCost, int Nverts,
                        int Nsets, int *AdjMat, VERTEX *Vertex, SET *Set) →
                        Int, NodeBestCost, Vertex, Set
end SCPNDCG
```

**ADT B.22**

```
structure SCPNDLB
      declare
            main(int argc, char *argv[], int Nverts, int Nsets, int *AdjMat,
                  VERTEX *Vertex, SET *Set) → Nverts, Nsets, AdjMat, Vertex, Set
            ScpSearch(TABLE_NODE *SetsToSearch, int *NodeBestCost, int Nverts,
                        int Nsets, int *AdjMat, VERTEX *Vertex, SET *Set) →
                        Int, NodeBestCost, Vertex, Set
            LoadBalance(S_NODE *SearchStack)
end SCPNDLB
```

**ADT B.23**

```
structure SCPNDFG
      declare
            main(int argc, char *argv[], int Nverts, int Nsets, int *AdjMat,
                  VERTEX *Vertex, SET *Set) → Nverts, Nsets, AdjMat, Vertex, Set
            ScpSearch(TABLE_NODE *SetsToSearch, int *NodeBestCost, int Nverts,
                        int Nsets, int *AdjMat, VERTEX *Vertex, SET *Set) →
                        Int, NodeBestCost, Vertex, Set
end SCPNDFG
```

**ADT B.24**

```
structure SRLSORTS
      declare
            MergeSort(char *Base_i, char *Base_j, int NumElements_i, int NumElements_j,
                        int ElementWidth, int (*cmpf)()) → Base_i, Base_j
            QuickSort(char *Base, int NumElements, int ElementWidth,
                        int (*cmpf)()) → Base
            InsertionSort(char *Base, int NumElements, int ElementWidth,
                        int (*cmpf)()) → Base
            SwapByte(char *aptr, char *bprt, int count)
end SRLSORTS
```

**ADT B.25**

```
structure SSCPNODE
      declare
            main(int argc, char *argv[], int Nverts, int Nsets, int *AdjMat,
                  VERTEX *Vertex, SET *Set) → Vertex, Set
            SScpSearch(TABLE_NODE *Table, int *CostOfCoveringSets,
                        int **CoveringSets, LZI_NODE **L,
                        unsigned long *BestCoverTime, int Nverts, int Nsets,
                        int *AdjMat, VERTEX *Vertex, SET *Set) → Int,
                        CostOfCoveringSets, CoveringSets, L, BestCoverTime,
                        Vertex, Set
end SSCPNODE
```

**ADT B.26**

```
structure STACK
      declare
            OpenStack(S_NODE **TopPtr) → TopPtr
            FlushStack(S_NODE **TopPtr) → TopPtr
            PushOnStack(S_NODE **TopPtr, ITEM *NewItemPtr) → TopPtr
            PopOffStack(S_NODE **TopPtr) → ITEM, TopPtr
            ItemsInStack(S_NODE *TopPtr) → Int
end STACK
```

## ADT B.27

structure STARTSET
    declare
        BuildStartingSets(TABLE_NODE *Table, Q_NODE **Front, Q_NODE **Rear,
                Q_NODE **Current, VERTEX *Vertex, SET *Set) →
                Int, Front, Rear, Current, Set
        NodeStartingSets(int NumExpandedLists, int NumSearchers, int MyNID) → Int
        NextStartingSet(Q_NODE *FrontPtr, Q_NODE **CurrentPtr, int NumSearchers,
                int MyNID) → TABLE_NODE, CurrentPtr
        BuildInitialSets(TABLE_NODE *Table, int NumSearchers, Q_NODE **QFront,
                Q_NODE **QRear, VERTEX *Vertex, SET *Set) →
                BOOLEAN, QFront, QRear, Vertex
        SendAnotherSet(int RequestingNID, Q_NODE **QFront,
                Q_NODE **QRear) → TABLE_NODE, QFront, QRear
    end STARTSET

## ADT B.28

structure TABLE
    declare
        BuildTable(TABLE_NODE **Table, int Nverts, int Nsets, int *AdjMat,
                VERTEX *Vertex, SET *Set) → BOOLEAN, Table
        BuildTableSegment(int *ExpandedSet, int Elements_ES,
                TABLE_NODE *Table) → TABLE_NODE
        PrintTable(TABLE_NODE *Table, int Nverts, int Nsets, int *AdjMat,
                VERTEX *Vertex, SET *Set)
        PrintTablePointers(TABLE_NODE *Table)
end TABLE

# Appendix C. *SCP User's and Programmer's Manual*

## Table of Contents



### *C.1 Introduction*

The SCP is the problem of finding the minimum number of columns in a 0–1 matrix[1] such that all rows of the matrix are covered by at least one element from any column and the cost associated with the covering columns is optimal (minimum or maximum) (17:39). As an example, Figure C.1 shows a 0–1 matrix in which the rows are covered by several different combinations of columns. Columns 0, 1, 2, 3, and 4 form a cover with a total cost of 27. The optimal cover is columns 0, 3, and 4 with a cost of 15.

One serial and three parallel versions of the SCP algorithm are implemented for the iPSC/2 hypercube. The serial SCP is treated as a separate entity and is stored, compiled,

[1] A 0–1 matrix is a rectangular matrix in which a covered row is denoted by a '1' in the covering columns. If the rows in the matrix represent the vertices of a graph, the existence of an arc between any two vertices is denoted by a '1' in the column of the matrix.

| | | Sets | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| Vertices | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 |
| | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 |
| | 2 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| | 3 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 1 |
| | 4 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 |
| | 5 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 |
| | | 4 | 7 | 5 | 8 | 3 | 2 | 6 | 5 |
| | | Costs | | | | | | | |

Figure C.1. 0–1 Matrix for a Set Covering Problem (17:54)

and executed from a separate directory; whereas, all three parallel programs are contained in the same directory and are invoked from a single host program. The following sections describe the structure of the serial and parallel programs, the input file structure, the output statistics, potentially reusable software, and suggested extensions to the programs.

## *C.2 Structure of Serial SCP Program*

As previously stated, all the programs required to compile and execute the serial programs are contained in a separate directory. A **makefile** is included to compile and link the various files. The end result of executing **make** on the files is an executable file called **scp**. The serial SCP options are explained in the following paragraph; Appendix A list the structure charts and the abstract data types.

A help screen, Figure C.2, is available and is displayed by typing **scp** with no other arguments. As shown in the figure, the SCP program is executed by typing **scp** followed by any options and the input filename. The structure of the input file is described in Section C.4 and the options follow:

**r** — Enables all three reduction techniques: Options 2, 3, and 4. See Sections E.6.4 and 4.5 for a full explanation.

**2** — Removes any rows that are covered by only one column. The removed column is saved and added to the solution upon completion of the search. If this option is

```
SERIAL SET COVERING PROBLEM (SCP) HELP SCREEN

>scp [r234dlats] File

Options available:
  r - Enable all reductions.
  2 - Enable reduction #2.
  3 - Enable reduction #3.
  4 - Enable reduction #4.
  d - Enable dominance testing.
  l - Enable lower bound testing.
  a - Print the 0-1 matrix.
  t - Print the table.
  s - Save the reduced/reordered 0-1 matrix to a file.
```

Figure C.2. Serial SCP Help Screen

chosen it is not possible to save (option **s**) the information contained in the original 0–1 input matrix.

**3** — Removes dominated rows. A is dominated if it is a superset of another row.

**4** — Removes dominated columns. A column is dominated if it is a subset of another row and has a higher cost.

**d** — Includes a dominance test in the search, reference Sections E.6.6 and 4.3. May slow the algorithm.

**l** — Includes a lower bound test in the search, reference Sections E.6.7 and 4.4. May slow the algorithm.

**a** — Causes the input 0–1 matrix to displayed in its original form and its reordered/reduced form.

**t** — Causes the table to be displayed, reference pages E-12, 3-8– 3-15, and 4-7.

**s** — If specified, allows the user to save the 0–1 matrix. If the filename exists the program asks for confirmation before proceeding. The user can tell the program to abort the write (**n**), overwrite the file (**y**), or specify a new filename (**f**).

## *C.3 Structure of Parallel SCP Programs*

Three parallel algorithms are implemented and integrated through a common host algorithm. The host program is called **scp** and the selection of the algorithm is accomplished on the command line by specifying either `C`, `F`, or `D`. As with the serial SCP, a help screen, Figure C.3, is available and is displayed by typing **scp** with no other arguments. The options are explained below and Appendix B contains the structure charts and abstract data types.

```
  PARALLEL SET COVERING PROBLEM (SCP) HELP SCREEN

>scp [Algorithm|Options] Searchers File

Algorithm|Options:
  C - Runs coarse grain SCP program -- GOOD.
  F - Runs fine grain SCP program -- BETTER.
  D - Runs dynamic load balanced SCP program -- BEST (default).

  S - Force a serial sort in the parallel search.
  P - Force a parallel sort in the parallel search.
  r - Enable all reductions.
  2 - Enable reduction #2.
  3 - Enable reduction #3.
  4 - Enable reduction #4.
  d - Enable dominance testing.
  l - Enable lower bound testing.
  n - Print individual node statistics.
  a - Print the 0-1 matrix.
  t - Print the table.
  s - Save the reduced/reordered 0-1 matrix to a file.

Searchers:
  Number of searching processors = 0---7(default).
  0 - Serial SCP program on one node.
```

Figure C.3. Parallel SCP Help Screen

Notice that the help screen lists four more options and has an input argument called `Searchers`.

C — This option loads and executes the coarse grain parallel SCP algorithms (**scpcntcg**, **scpndcg**). See Section 4.2.3.1 for the algorithms.

F — This option loads and executes the fine grain parallel SCP algorithms (**scpcntfg**, **scpndfg**). This algorithm has a more efficient load balancing scheme than does the coarse grain algorithms; therefore, it should execute faster. See Section 4.2.3.2 for the algorithms.

D — This option loads and executes the dynamic load balanced parallel SCP algorithms (**scpcntlb**, **scpndlb**, **lbproc**). These algorithms are similar to the fine grain algorithms except that the capability to balance the load dynamically has been added. See Section 4.2.3.3 for the algorithms. Notice that this selection is the default; therefore, should the command line contain an entry similar to:

```
>scp filename
```

the host program will select the dynamic load balanced algorithms and the maximum number of processors available on the computer.

S — The normal operation of the sorting algorithms is to execute a parallel or serial sort depending on the number of elements to be sorted. This option forces the parallel SCP algorithms to sort the elements using a serial quick sort or insertion sort.

P — Forces the parallel SCP algorithms to execute a parallel bitonic merge sort.

r — Enables all three serial or parallel reduction techniques: Options 2, 3, and 4. If the specified SCP search algorithm is parallel, then the reductions are parallel, else the reductions are serial. See Sections E.6.4 and 4.5 for a full explanation of the serial and parallel reduction algorithms.

2 — Removes any rows that are covered by only one column. The removed column is saved and added to the solution upon completion of the search. If this option is chosen it is not possible to save (option **s**) the information contained in the original 0-1 input matrix. Always a serial algorithm.

3 — Removes dominated rows. A is dominated if it is a superset of another row. May be parallel or serial.

4 — Removes dominated columns. A column is dominated if it is a subset of another row and has a higher cost. May be parallel or serial.

d — Includes a dominance test in the search, reference Sections E.6.6 and 4.3. May slow the algorithm.

l — Includes a lower bound test in the search, reference Sections E.6.7 and 4.4. May slow the algorithm.

n — This option displays the searching processor statistics in addition to the default display which only shows the run-time statistics for the controlling processor.

a — Causes the input 0–1 matrix to displayed in its original form and its reordered/reduced form.

t — Causes the table to be displayed, reference pages E-12, 3-8– 3-15, and 4-7.

s — If specified, allows the user to save the 0–1 matrix. If the filename exists the program asks for confirmation before proceeding. The user can tell the program to abort the write (**n**), overwrite the file **y**), or specify a new filename (**f**).

The **Searchers** argument specifies the number of searching processors. The user may select any number from '0' to the one less than the maximum number of nodes contained on the iPSC/2. Notice that selecting '0' searchers loads and executes a serial program. This option is included allow the user to run-time statistics for a serial SCP algorithm executed on one of the node processors. In most instances, the execution times are shorter on a node than they are on the host because no context switch time is incurred on a node processor. Furthermore, since the node processor is not shared with other users or processes, the programs require less real-time to finish.

Should the user select more searching processors than are currently available, the SCP program prints the following message (# is the number of nodes):

```
# node(s) not available, retrying ...
```

and then enters a loop where it continually waits ten seconds and tries again to acquire the proper number of nodes. The program does not exit this loop until it gets the nodes or the user aborts the program. Remember that the number of nodes acquired is always $2^d$; therefore, if the user selects four searchers, a cube of dimension 3 (i.e., eight nodes) is required to execute the programs with four searching processors and one controlling processor.

### *C.4 Structure of Input 0–1 Matrix*

The 0–1 matrix is entered into the program via a file. The input file must include the number of rows and columns, the matrix proper, and the column costs. Figure C.4 is a typical input matrix. The first seven lines in the example are comments and are not

```
#
#           Filename: chris.dat
#           Number of Vertices: 6
#           Number of Sets: 8
#           Density of 1's: 0.416667
#           Cost range: [2,8]
#
6
8
11000010
10100101
00001110
00010000
01001011
11100101
4 7 5 8 3 2 6 5
```

Figure C.4. Input File Format for the SCP

required; however, they are added to any output file that the SCP programs write by specifying option **s** in the command line. The file input algorithm considers any line in the beginning of the file which starts with a # as a comment. The first line without a # ends the comment header and no other comments are recognized. The first line after the comments must be the number of rows and the second line is the number of columns in

the 0–1 matrix. The 0–1 matrix is entered in the following lines. The last line of the file contains the column costs ($> 0$) with each cost separated by a blank space.

### *C.5 Output Displays*

Upon completion of the search, the optimal solution, matrix data, and run-time statistics are displayed to the terminal. The matrix data consists of the input 0–1 matrix, the reordered/reduced 0–1 matrix, and the table; whereas, the run-time statistics consist of search time, total time, number of nodes expanded, and various other information depending on the option chosen and the particular program executed.

*C.5.1 0–1 Matrix Display* The 0–1 matrix is displayed if the user selects option **a** on the command line. Furthermore, the reduced/reordered 0–1 matrix displayed may appear different between consecutive runs on the same input data depending on whether any reductions (i.e., **r**, **2**, **3**, **4**) were chosen. The following display is the matrix for the data of Figure C.4 with options **3** and **4** chosen:

```
**************************************************************************
The original matrix(6X8):
Row 0)  11100101
Row 1)  10100101
Row 2)  00010000
Row 3)  01001011
Row 4)  00001110
Row 5)  11000010

Set Columns:
0 1 2 3 4 5 6 7

Set Costs:
4 7 5 8 3 2 6 5

**************************************************************************
Reduced matrix(4X5):
Row 1)  10101
Row 3)  01011
Row 4)  01110
Row 5)  10010
```

```
Set Columns:
0 4 5 6 7

Set Costs:
4 3 2 6 5

*****************************************************************************
Reordered matrix(4X5):
Row 5)  00101
Row 1)  10110
Row 3)  01011
Row 4)  11001

Set Columns:
5 4 0 7 6

Set Costs:
2 3 4 5 6
```

The original matrix matrix displays the input 0–1 matrix, its dimensions, and the column (set) costs. The reduced matrix is the 0–1 matrix after the specified reductions are finished. The reordered matrix is the 0–1 matrix following sorts on the rows and columns. The rows are sorted in ascending order according to the cardinality of the rows and the columns are sorted in ascending order according the their costs. In the case of the parallel algorithms, the reduced and reordered matrices are combined into one reordered/reduced matrix display.

*C.5.2 Table Display* A table is constructed for every search may be displayed by specifying option `t` on the command line. The displayed table, as shown, is merely a representation of the table constructed by the algorithms. The actual table consists of pointers into vectors which point to the 0–1 matrix as illustrated in Figure 3.1. Even so, the displayed table is an accurate representation of the internal data structures. The following table is for the reduced matrix shown previously:

```
**************************************************************************
The table:

Block 0 sets: (0, 6)
Block 0 set costs: (4, 6)
Block 1 sets: (5, 0, 7)
Block 1 set costs: (2, 4, 5)
Block 2 sets: (4, 7, 6)
Block 2 set costs: (3, 5, 6)
Block 3 sets: (5, 4, 6)
Block 3 set costs: (2, 3, 6)

Vertex 5)       11000000000
Vertex 1)       10111000000
Vertex 3)       01001111000
Vertex 4)       01100101111
```

*C.5.3 Optimal Solution* The optimal solution is displayed next. It consists of the covering sets and the cost. Note that the covering sets refer to the original 0–1 matrix. The following display segment is the solution to the previous example:

```
**************************************************************************
Covering Sets are:
( 0 4 3 )
Cost = 15
```

The search graph for this optimal cover is shown in Figure C.5 and is based on a search of the table in Section C.5. Since the table represents the reduced matrix, it does not contain column 3. This column is removed by Reduction #2 and is later added to the final solution of the reduced matrix. The search proceeds in the following manner:

1. Select column 0 from the table and added to the list of covering columns.
2. Select column 4 from the table and added to the list of covering columns.
3. A cover of the rows exists and it is less than the best cover found so far; therefore, save the list of covering columns and the cost.
4. Backtrack from node 4 to node 0.

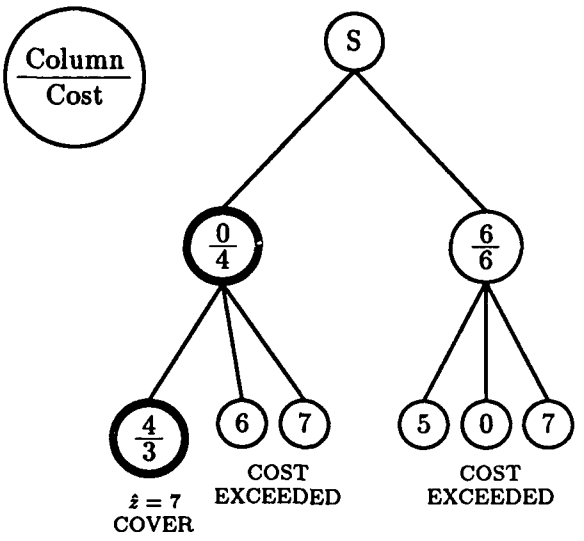



Figure C.5. Serial Search Graph for the Input Matrix of Figure C.4

5. From node 0, consider column 7 for a covering column. The addition of column 7 can not result in a lower cost cover; therefore, try next column.
6. From node 0, consider column 6 for a covering column. The addition of column 6 can not result in a lower cost cover and all columns have been exhausted; therefore, backtrack to node S.
7. Select column 6 from the table and add to the list of covering columns.
8. Consider the column 5 from the table. The addition of column 5 will exceed the best cost obtained thus far; therefore, it can not lead to a better solution.
9. Same argument holds for columns 0 and 7.
10. Search is complete, exit with the optimal cover consisting of columns 0 and 4 for a summed cost of 7.
11. Add column 3 removed by reduction #2 to the optimal solution. Optimal solution is now columns {0, 4, 3} for a total cost of 15.

*C.5.4 Output Statistics* The final portion of the display consists of the run-time statistics. Each version of the algorithms includes slightly different information in its

display. The times presented in the displays are CPU time for the serial program executing on the host and wall-clock time for all versions of the parallel programs executing on the node processors (35:3-53).

*Serial Algorithm Statistics* This section shows the output display from the serial SCP program which executes on the host processor.

```
**************************************************************************
The Statistics:
Time to reduce matrix = 0.020000 seconds.
Time to sort = 0.010000 seconds.
Time to build the table = 0.010000 seconds.
Time to search the tree = 0.010000 seconds.
Expanded 3 nodes of the search tree.
Total time  = 0.050000 seconds.
```

**Time to reduce matrix** — This is the cumulative time to apply all user requested reductions to the 0–1 matrix.

**Time to sort** — Cumulative time to sort the `Vertex` and `Set` vectors.

**Time to build the table** — Time required to build the internal representation of the table.

**Time to search the tree** — The time for the serial program to search the tree. Does not include any of the above mentioned times.

**Expanded ? nodes of the search tree** — The number of nodes expanded in the search tree.

**Total time** — Total CPU time to execute the SCP program including all the above times.

*Parallel Algorithm Statistics* Each processor in the the parallel algorithms searches a separate portion of the search tree. Consider a parallel search of the search graph in Figure C.5 by two searching processors. Processor 1 is instructed to search the left subbranch beginning with node 0 and Processor 2 is instructed to search the right subbranch beginning at node 6. The optimal solution for this example does not change.

The important component of the optimal solution is the cost. The list of covering sets may vary, but the optimal cost can not change.

The parallel algorithms always display the run-time statistics for the control processor and will display statistics from each searching processor if option **n** is entered on the command line. The following displays show the statistics for options **r**, **n**, **a**, **t**, and one searching processor for each of the three parallel algorithms. Rather than repeat redundant information, a description of all the coarse grain algorithm statistics is given and then only those statistics which change or are new are described for the fine grain and dynamic load balanced algorithms:

```
**************************************************************************
Coarse Grain Program Statistics:

NODE 1 Stats:
  Time to build table = 0.007 seconds.
  Time to build the subgraphs = 0.003 seconds.
  Time spent searching = 0.003 seconds.
  Number of subgraphs searched = 10.
  Expanded 0 nodes of the search tree.
  Total processor time = 0.038 seconds.

Control Node (Node 0) Stats:
  Time to reduce (4X5) = 0.011 seconds.
  Time to sort = 0.008 seconds.
  Time to search the tree = 0.014 seconds.
  Time until best cover was first found = 0.013 seconds.
  Total time = 0.042 seconds.

  Sent 3 global costs to the searching processors.
  Total number of expanded nodes = 0.
  Overall search efficiency (0.003/(1*0.042)) = 7.1%.
```

**Time to build the table** — Time required to build the internal representation of the table.

**Time to build the subgraphs** — This is the time to build the subgraphs so that each searching processor receives at least one subgraph.

**Time spent searching** — Time spent searching for a cover. Does not include any other time (i.e., time to build the table, reduce).

**Number of subgraphs searched** — The number of subgraphs searched from the list of subgraphs.

**Expanded 0 nodes of the search tree** — The number of search tree nodes expanded by the processor. Does not include the nodes already expanded in the subgraph.

**Total processor time** — Total time that the processor worked on the search. Includes all other times.

**Time to reduce** — This is the cumulative time to apply all user requested reductions to the 0–1 matrix. May be serial or parallel reductions.

**Time to sort** — Cumulative time to sort the `Vertex` and `Set` vectors. May be a serial or parallel sorting algorithm.

**Time to search the tree** — Time from when the controller starts the search until the last searching processor terminates.

**Time until best cover was first found** — Time from the beginning of the search until the optimal cover was sent to the controller. Does not include the time required to finish the search.

**Total time** — Total time on the controller to receive the base data structures and to complete the search.

**Sent 3 global costs** — The number of times the controller sent new global best costs to the searching processors.

**Total number of expanded nodes** — Total nodes expanded by all searching processors.

**Overall search efficiency** — Sum of the processor search times divided by product of the number of searchers ($m$) and the total time:

$$\frac{\sum_{i=1}^{m} \text{SearchTime}}{m \times \text{TotalTime}}$$

```
*************************************************************************
Fine Grain Program Statistics:

NODE 1 Stats:
  Time to build table = 0.008 seconds.
  Time spent waiting for subgraphs to search = 0.021 seconds.
  Time spent searching = 0.002 seconds.
  Search efficiency (0.002/0.023) = 8.7%.
  Number of subgraphs searched = 5.
  Expanded 1 nodes of the search tree.
  Total processor time = 0.054 seconds.

Control Node (Node 0) Stats:
  Time to reduce (4X5) = 0.010 seconds.
  Time to sort = 0.008 seconds.
  Time to build the table = 0.001 seconds.
  Time to search the tree = 0.031 seconds.
  Time until best cover was first found = 0.030 seconds.
  Total time = 0.059 seconds.

  Sent 2 global costs to the searching processors.
  Total number of expanded nodes = 1.
  Overall search efficiency (0.002/(1*0.059)) = 3.4%.
```

**Time spent waiting for subgraphs to search** — Since the controller is now expanding the subgraphs, the searchers must wait for a subgraph to arrive before starting the search algorithm. This time is the cumulative time the searching processor spent waiting for subgraphs either at the beginning of the search or after requesting a another subgraph.

**Search efficiency** — Search time divided by the time spent searching and waiting for subgraphs:

```
*************************************************************************
Dynamic Load Balanced Program Statistics:

NODE 1 Stats:
  Time to build table = 0.008 seconds.
  Time spent waiting for subgraphs to search = 0.045 seconds.
  Time spent doing dynamic load balancing = 0.000 seconds.
```

```
  Time spent searching = 0.003 seconds.
  Search efficiency (0.007/0.031) = 6.3%.
  Searched a total of 3 subgraphs, 0 from load balancing.
  Expanded 4 nodes of the search tree, 0 from load balancing.
  Total processor time = 0.125 seconds.

Control Node (Node 0) Stats:
  Time to reduce (4X5) = 0.011 seconds.
  Time to sort = 0.008 seconds.
  Time to build the table = 0.001 seconds.
  Time to search the tree = 0.057 seconds.
  Time until best cover was first found = 0.010 seconds.
  Time when dynamic load balance started = 0.035 seconds.
  Total time = 0.165 seconds.

  Sent 1 global costs to the searching processors.
  Total number of expanded nodes = 4.
  Overall search efficiency (0.007/(1*0.142)) = 1.8%.
```

**Time spent waiting for subgraphs to search** — Includes the time spent waiting for subgraphs from the controller and other searching processors as a result of a request for a subgraph during the dynamic load balancing phase of the algorithm.

**Time spent doing dynamic load balancing** — Time spent breaking off subgraphs to share with other searching processors.

**Time spent searching** — Does not include the time spent doing dynamic load balancing or waiting for subgraphs.

**Time when dynamic load balance started** — Time when the token was fired by the controller.

### *C.6 Reusable Software*

The construction of reusable parallel programs is difficult at best; however, the algorithms presented here are adaptable to many programs executing on an iPSC/2.

*C.6.1 Parallel Bitonic Merge Sort* The parallel sort employed in the SCP is an enhanced version of a bitonic merge sort obtained from Quinn (50:93–94). Quinn's version of the algorithm assumes $2^d$ items are to be sorted in ascending order on $2^d$ processors.

His algorithm is modified to allow any size input data to be sorted in either ascending or descending order on $2^d$ processors, reference Section 4.6 of this thesis for the algorithm. To invoke the parallel sort, each processor executes a command with the following prototype:

```
int ParSort(int Order, char *Base, int NumElements, int ElementWidth,
            int (*cmpf)(), int (*apadf)(), int (*rpadf)(), int CubeDim);
```

where **Order** is **ASCENDING** or **DESCENDING**, **Base** is a pointer to the beginning of the data to be sorted, **NumElements** is the number of items pointed to by **Base**, **ElementWidth** is the size of the items pointed to by **Base**, **(*cmpf)()** is a user supplied function to compare two items, **(*apadf)()** is a user supplied function to add a padding item to the data, **(*rpadf)()** is a user supplied function to remove a padding item from the sorted data, and **CubeDim** is the dimension of the cube.

The bitonic merge must execute on all $2^d$ processors in a synchronous manner. In order to accomplish this, a single processor divides the data between all processors (including itself) and then enters the sort algorithm along with the other processors. If the number of items to be sorted is not divisible by $2^d$, then some of the processors must pad their received data set because they have received one less item than everyone else. The routine to pad the input data must be supplied by the user since the pad value depends on the type of data to be sorted. For example, the SCP programs sort the **Vertex** vector in ascending order based on the cardinality of the rows. The add pad function simply inserts a **−MAXINT** in the cardinality field of the record succeeding the last **Vertex** record received if the processor received one of the smaller **Vertex** vectors. When the records are finally sorted and collected by the controlling processor, all the padding is at the front of the slightly enlarged **Vertex** vector. A user supplied function is then employed to remove all padding at from the vector leaving the properly sorted **Vertex** vector.

When all data is partitioned to the processors and any necessary padding has been added, all processors execute the bitonic merge sort. Each processor must first sort its input list of data; hence, a quick sort is executed. Following the quick sort, the processors send and receive lists according to the algorithm. Since, the quick sort is generic, it requires a user supplied function (**cmpf**) to compare the individual elements. **Cmpf** must return a

'1' if two input items are out of order, a '0' if they are equal, and a '−1' if they are in order. For the case of the **Vertex** vector, **cmpf** compares the cardinality fields of two input records.

A file containing examples routines to sort a list of integers is provided in the directory with the parallel SCP algorithms under **/Sort/bmerge.ex**. In order to call this program, each processor must execute the following command:

```
int ParSort(ASCENDING, (char *)List, NumElements, sizeof(int),
            Ascending, Pad, Unpad, CubeDim);
```

Where **List** is the list of integers to be sorted and **CubeDim** is the dimension of the cube.

*C.6.2 Dynamic Load Balancing Algorithm* As discussed in Section 4.2.3.3, the fine grain algorithm performs better than the coarse grain algorithm but searching processors still exhibit relatively long idle periods. The idle periods are a result of a load imbalance since the finished searchers are waiting for all searchers to terminate. The addition of a dynamic load balancing algorithm to the fine grain algorithm decreases the processor idle-times and leads to a decrease in the overall solution time.

The dynamic load balancing algorithm consists of four logical units. The major unit is a load balancing algorithm which exists as a separate process on the controller and all searchers. This algorithm is solely responsible for coordinating the sharing of subgraphs. The other three logical units are interfaces to the load balancing process. One interface is required on the controller to initiate the dynamic load balancing algorithm; whereas, two interfaces are required on the searchers to request and share subgraphs.

The searchers request subgraphs from the controller until the controller's list of subgraphs is depleted. When a searcher requests another subgraph and is informed that the controller has no more subgraphs, it requests a subgraph from its load balancing process. The load balancing process waits for the token to arrive and then polls working processors for a subgraph. If it receives a subgraph, it passes the subgraph to the searcher and then passes the token to the next processor. Notice that the searching processes do not

communicate with each other. The load balancing process coordinates intrasearcher communications and subgraph sharing and it is allowed to communicate with other searchers only when it has the token. These two restrictions on the dynamic load balancing scheme prevent deadlock and a race condition from occurring.

The final interface between a searching process and a load balancing process shares a subgraph. When a load balancing process receives a request for a subgraph, it immediately requests the subgraph from its searching process. The searching process must interrupt its search process and partition its subgraph. The searcher then sends a subgraph, or an empty subgraph, to the load balancing process which relays the subgraph to the requesting load balancing process.

The algorithms for the dynamic load balancing algorithm are in Section 4.2.3.3. Code segments are available in the directory containing the parallel source code in a directory called `LoadBal`; whereas, SCP routines `scpcntlb.c`, `scpndlb.c`, and `lbproc.c` contain the dynamic load balancing algorithms.

### *C.7 Program Extensions*

The following extensions are suggested:

- The column costs in the input matrix must be greater than zero. Typically, this is not a problem; however, it is conceivable that a user may want to search matrices with zero costs. To add this capability to the programs requires modifications in not only the search routines, but also the reduction, dominance test, and lower bound test routines.
- The maximum number of processors displayed on the parallel SCP help screen is hard coded in `scpgbl.h`. A convenient system variable is not available to read the value; therefore, if the size of the iPSC/2 increases, `MAX_DIM` in `scpgbl.h` must be changed and the programs recompiled.

# Appendix D. *Raw Test Result Data*

## List of Tables



This appendix contains the raw test results from the execution of the serial and parallel SCP programs. The actual printouts from the tests are not given since they are quite large; however, they are available in directory `/results`.

**Table D.1** — Shows the optimal solution and a corresponding list of covering sets to the twenty-nine test problems. These solutions were obtained from the serial SCP algorithm executed on AFIT's hypercube. The names of the matrices indicate the contents of the problem. For instance, matrix 100.100.49 is a 100 × 100 matrix with a 1's density of 0.49. Matrix 75.125.25.*U* is a 75 × 125 matrix with a 1's density of 0.25 and unit cost columns. Matrix 100.100.30.v1 is version 1 of a 100 × 100 matrix with a 1's density of 0.30. In addition to the naming scheme, the beginning of these problems contain a header (ref. Figure C.4) to indicate the name, size, and density of the matrix.

**Table D.2** — Execution times for the sorting and reduction algorithms. The serial times are the algorithms executed by the serial SCP program and the parallel times are from the dynamic load balanced parallel SCP algorithm executed on three different cube dimensions since the parallel sort and reduction algorithms require $2^d$ processors

where $d$ is the dimension of the cube. The serial sort algorithm is a quick sort and the parallel sort is a bitonic merge sort. These times were collected at AFIT.

**Table D.3** — The maximum time that a searching processor was idle during a search of the test matrix. This time is the maximum obtained for any number of processors. The data is obtained from Intel's hypercube.

**Table D.4** — A serial SCP program is available which executes on a node processor rather than the host processor. This program is a strict serial program and it contains no parallel algorithms. It is executed on a node processor because the node processors are single user; hence, all CPU time is dedicated to the single user and the program finishes sooner. This table shows the elapsed time from the beginning of the search until the best cost was last updated (BCT), the search time (ST), the total execution time (TT), and the number of search tree nodes expanded (EN). Only the solution times for the five most time consuming test matrices is shown since the rest of the matrices are solved much too quickly for use in comparing the serial and parallel search algorithms. These times are the base times from which the speedup is calculated were obtained from Intel's hypercube.

**Tables D.5, D.6, and D.7** — These tables contain the same four measurements shown in Table D.4 (i.e., BCT, ST, TT, and EN) for the same five test matrices searched on Intel's hypercube. The **searchers** column refers to the number of processors searching the input matrix and ranges from 1 to 31 searching processors.

**Table D.8** — The speedup obtained when searching one of the five test matrices for 1–31 searching processors. The speedups are computed by dividing the times displayed in Table D.4 by the search times (ST) displayed in Tables D.5, D.6, and D.7.

Table D.1. Test Matrix Solutions

| Matrix | Cost | Covering Sets |
|---|---|---|
| 10.10.53 | 322 | {8 0} |
| 100.100.26.U | 6 | {0 4 52 30 44 36} |
| 100.100.27.U | 6 | {4 87 3 28 92 44} |
| 100.100.28.U | 6 | {2 6 36 38 3 82} |
| 100.100.30.v1 | 629 | {76 32 67 29 90 96 45} |
| 100.100.30.v2 | 684 | {46 38 36 6 71 86 52 34} |
| 100.100.30.v3 | 540 | {18 50 85 79 35 99 24 94 51} |
| 100.100.35.v1 | 500 | {38 0 22 80 83 56 3} |
| 100.100.35.v2 | 381 | {13 51 27 54 56 84 10 96} |
| 100.100.35.v3 | 496 | {0 60 59 25 9 79 47} |
| 100.100.40.v1 | 369 | {62 36 22 29 31 66 52} |
| 100.100.40.v2 | 478 | {81 39 97 49 57 64} |
| 100.100.40.v3 | 258 | {31 62 97 26 5 11} |
| 100.100.49 | 67 | {32 15 91 90 84} |
| 100.100.50.v1 | 133 | {70 87 80 0 75 92} |
| 100.100.50.v2 | 425 | {80 26 3 67 92} |
| 100.100.50.v3 | 311 | {81 49 36 94} |
| 20.20.05 | 3058 | {0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19} |
| 20.20.43 | 503 | {6 14 16 1 19} |
| 20.20.46 | 482 | {6 14 16 19} |
| 20.20.51 | 50 | {6} |
| 20.20.54 | 75 | {1 17 19} |
| 30.30.45 | 107 | {3 24 17 10} |
| 40.40.41 | 164 | {25 14 38 12} |
| 50.50.52 | 149 | {30 41 36 46} |
| 70.70.08.U | 16 | {29 38 21 27 9 45 24 5 8 56 42 0 54 65 37 44} |
| 75.125.25.U | 5 | {116 47 10 69 101} |
| 75.75.50 | 109 | {2 66 60 39 59} |
| chris | 15 | {3 4 0} |

Table D.2. Support Times for Matrix 1000.700.50 (seconds)

| Function | Serial | Parallel | | |
|---|---|---|---|---|
| | | 2 | 4 | 8 |
| Sort | 0.49 | 0.35 | 0.23 | 0.22 |
| Reduce | 55.95 | 39.21 | 36.45 | 37.32 |

Table D.3. Maximum Searcher Idle-Time (seconds)

| Matrix | Coarse Grain | Fine Grain | Load Balance |
|---|---|---|---|
| 100.100.28.U | 1065 | 395 | 42 |
| 100.100.27.U | 678 | 357 | 40 |
| 100.100.26.U | 348 | 329 | 33 |
| 75.125.25.U | 85 | 152 | 19 |
| 70.70.08.U | 198 | 1444 | 42 |

Table D.4. Serial Node Statistics

| Matrix | BCT | ST | TT | EN |
|---|---|---|---|---|
| 100.100.28.U | 100.4 | 6925.5 | 6925.8 | 3479327 |
| 100.100.27.U | 5142.7 | 10943.8 | 10944.2 | 5578362 |
| 100.100.26.U | 228.5 | 5578.5 | 5578.8 | 2806938 |
| 75.125.25.U | 9351.3 | 9362.9 | 9363.2 | 6264322 |
| 70.70.08.U | 2010.2 | 4443.3 | 4443.4 | 3877337 |

BCT: Time when best cover was initially found.
ST: Search time.
TT: Total program execution time.
EN: Number of expanded nodes
Time is in seconds.

Table D.5. Coarse Grain Parallel SCP Statistics

| Searchers | 100.100.28.U | | | | 100.100.27.U | | | |
|---|---|---|---|---|---|---|---|---|
| | BCT | ST | TT | EN | BCT | ST | TT | EN |
| 1 | 32.7 | 19090.7 | 19090.9 | 10797313 | 4403.3 | 21004.5 | 21004.7 | 11765704 |
| 2 | 32.8 | 9751.0 | 9751.2 | 10815454 | 927.5 | 9293.4 | 9293.6 | 10378245 |
| 3 | 32.7 | 6633.3 | 6633.5 | 10833420 | 903.5 | 6497.2 | 6497.4 | 10830299 |
| 4 | 31.2 | 5100.9 | 5101.2 | 10847710 | 903.4 | 5107.6 | 5107.8 | 11326037 |
| 5 | 0.5 | 3927.5 | 3927.8 | 10779589 | 902.9 | 4640.0 | 4640.3 | 11827011 |
| 6 | 0.5 | 3457.3 | 3457.5 | 10779696 | 902.9 | 3679.0 | 3679.3 | 12306503 |
| 7 | 0.5 | 3117.4 | 3117.6 | 10779805 | 902.9 | 3626.1 | 3626.3 | 12785815 |
| 8 | 0.5 | 2742.5 | 2742.7 | 10779905 | 902.9 | 3286.5 | 3286.7 | 13284606 |
| 9 | 0.5 | 2334.5 | 2334.7 | 10780009 | 389.4 | 2444.9 | 2445.1 | 11267453 |
| 10 | 0.5 | 2268.1 | 2268.4 | 10780130 | 389.4 | 2425.4 | 2425.7 | 11474401 |
| 11 | 0.5 | 1970.5 | 1970.8 | 10780242 | 389.4 | 2408.4 | 2408.7 | 11685007 |
| 12 | 0.5 | 2269.5 | 2269.7 | 10780351 | 387.5 | 1803.2 | 1803.4 | 11891900 |
| 13 | 0.5 | 1908.9 | 1909.2 | 10780448 | 371.7 | 1767.2 | 1767.5 | 11988161 |
| 14 | 0.5 | 1583.8 | 1584.0 | 10780552 | 387.5 | 1796.2 | 1796.4 | 12312357 |
| 15 | 0.5 | 1565.9 | 1566.1 | 10780649 | 386.5 | 1771.1 | 1771.4 | 12517949 |
| 16 | 0.5 | 1576.2 | 1576.4 | 10780747 | 387.5 | 1730.2 | 1730.4 | 12739724 |
| 17 | 0.5 | 1507.4 | 1507.7 | 10780858 | 379.8 | 1432.4 | 1432.7 | 12883412 |
| 18 | 0.5 | 1186.2 | 1186.5 | 10780941 | 387.5 | 1439.6 | 1439.8 | 13166684 |
| 19 | 0.5 | 1185.5 | 1185.8 | 10781052 | 387.5 | 1440.0 | 1440.2 | 13379122 |
| 20 | 0.5 | 1186.3 | 1186.5 | 10781140 | 384.9 | 1437.5 | 1437.8 | 13574698 |
| 21 | 0.5 | 1187.0 | 1187.3 | 10781240 | 384.9 | 1400.1 | 1400.4 | 13782799 |
| 22 | 0.5 | 1159.7 | 1160.0 | 10781336 | 384.9 | 1397.2 | 1397.4 | 13994800 |
| 23 | 0.5 | 1169.1 | 1169.3 | 10781430 | 384.9 | 1399.2 | 1399.5 | 14202394 |
| 24 | 0.5 | 1168.6 | 1168.9 | 10781525 | 384.9 | 1409.8 | 1410.1 | 14411799 |
| 25 | 0.5 | 1115.7 | 1116.0 | 10781619 | 384.9 | 1393.9 | 1394.1 | 14625957 |
| 26 | 0.5 | 1115.7 | 1116.0 | 10781713 | 27.5 | 1052.8 | 1053.0 | 9759962 |
| 27 | 0.5 | 1114.0 | 1114.3 | 10781808 | 27.5 | 1064.1 | 1064.4 | 9774623 |
| 28 | 0.5 | 794.9 | 795.2 | 10781904 | 27.5 | 1064.2 | 1064.5 | 9789384 |
| 29 | 0.5 | 1101.2 | 1101.5 | 10781994 | 27.5 | 1064.2 | 1064.5 | 9804006 |
| 30 | 0.5 | 1115.2 | 1115.5 | 10782092 | 27.5 | 1064.2 | 1064.4 | 9819343 |
| 31 | 0.5 | 1115.7 | 1116.0 | 10782175 | 27.5 | 1064.6 | 1064.8 | 9834373 |

BCT: Time when best cover was initially found.
ST: Search time.
TT: Total program execution time.
EN: Number of expanded nodes
Time is in seconds.

Table D.5 (Continued). Coarse Grain Parallel SCP Statistics

| Searchers | 100.100.26.U | | | | 75.125.25.U | | | |
|---|---|---|---|---|---|---|---|---|
| | BCT | ST | TT | EN | BCT | ST | TT | EN |
| 1 | 2950.9 | 7636.7 | 7636.9 | 4295730 | 7612.3 | 7715.0 | 7715.3 | 5670010 |
| 2 | 2831.5 | 5115.0 | 5115.2 | 5783216 | 3491.3 | 3548.1 | 3548.3 | 5196359 |
| 3 | 88.1 | 1735.1 | 1735.3 | 2838990 | 2728.6 | 2770.5 | 2770.8 | 6104401 |
| 4 | 88.1 | 1338.5 | 1338.8 | 2888152 | 1922.4 | 1955.8 | 1956.0 | 5717210 |
| 5 | 88.1 | 1237.2 | 1237.5 | 2939072 | 1522.5 | 1552.8 | 1553.1 | 5693529 |
| 6 | 88.1 | 920.3 | 920.5 | 2988141 | 963.5 | 989.7 | 989.9 | 4269222 |
| 7 | 88.1 | 901.9 | 902.1 | 3038501 | 632.8 | 656.4 | 656.6 | 3337394 |
| 8 | 88.1 | 865.8 | 866.1 | 3088695 | 610.2 | 631.7 | 632.0 | 3656328 |
| 9 | 88.1 | 828.3 | 828.6 | 3136763 | 563.1 | 585.9 | 586.1 | 3788437 |
| 10 | 9.3 | 750.6 | 750.9 | 2738976 | 610.2 | 629.6 | 629.9 | 4508604 |
| 11 | 9.3 | 728.5 | 728.8 | 2744058 | 589.7 | 608.5 | 608.8 | 4779938 |
| 12 | 9.3 | 441.0 | 441.3 | 2749052 | 594.4 | 612.3 | 612.6 | 5212828 |
| 13 | 9.3 | 441.9 | 442.2 | 2754286 | 550.5 | 567.9 | 568.2 | 5244602 |
| 14 | 9.3 | 426.2 | 426.5 | 2759241 | 139.0 | 156.7 | 157.0 | 1577373 |
| 15 | 9.3 | 425.2 | 425.4 | 2764166 | 139.0 | 156.5 | 156.7 | 1673401 |
| 16 | 9.3 | 409.9 | 410.2 | 2769139 | 139.0 | 156.1 | 156.4 | 1770799 |
| 17 | 9.3 | 411.1 | 411.4 | 2774041 | 139.0 | 155.6 | 155.9 | 1866856 |
| 18 | 9.3 | 408.7 | 409.0 | 2779084 | 139.0 | 155.5 | 155.8 | 1919056 |
| 19 | 9.3 | 395.7 | 396.0 | 2784155 | 139.0 | 155.6 | 155.8 | 2015936 |
| 20 | 9.3 | 397.0 | 397.2 | 2789013 | 139.0 | 154.6 | 154.9 | 2116906 |
| 21 | 9.3 | 395.3 | 395.6 | 2793886 | 139.0 | 154.6 | 154.9 | 2109268 |
| 22 | 9.3 | 395.2 | 395.5 | 2799109 | 139.0 | 154.9 | 155.1 | 2156475 |
| 23 | 9.3 | 393.9 | 394.2 | 2804148 | 139.0 | 154.0 | 154.3 | 2221447 |
| 24 | 9.3 | 379.3 | 379.6 | 2809007 | 139.0 | 154.1 | 154.4 | 2260670 |
| 25 | 9.3 | 378.3 | 378.6 | 2813930 | 139.0 | 154.0 | 154.3 | 2267809 |
| 26 | 9.3 | 380.5 | 380.7 | 2818978 | 139.0 | 154.0 | 154.3 | 2308040 |
| 27 | 9.3 | 378.4 | 378.7 | 2824066 | 139.0 | 153.6 | 153.9 | 2362605 |
| 28 | 9.3 | 378.9 | 379.1 | 2829030 | 139.0 | 153.2 | 153.5 | 2426202 |
| 29 | 9.3 | 379.5 | 379.8 | 2834024 | 139.0 | 153.2 | 153.5 | 2437326 |
| 30 | 9.3 | 379.2 | 379.4 | 2838872 | 139.0 | 153.4 | 153.7 | 2443148 |
| 31 | 9.3 | 378.5 | 378.8 | 2844085 | 139.0 | 153.4 | 153.6 | 2452956 |

BCT: Time when best cover was initially found.
ST: Search time.
TT: Total program execution time.
EN: Number of expanded nodes
Time is in seconds.

Table D.5 (Continued). Coarse Grain Parallel SCP Statistics

| Searchers | 70.70.08.U | | | |
|---|---|---|---|---|
| | BCT | ST | TT | EN |
| 1 | 823.6 | 3453.9 | 3454.0 | 3233898 |
| 2 | 303.8 | 1637.9 | 1638.1 | 3039127 |
| 3 | 195.4 | 1161.4 | 1161.6 | 3035211 |
| 4 | 195.4 | 860.9 | 861.0 | 3178526 |
| 5 | 189.5 | 782.9 | 783.1 | 3300086 |
| 6 | 62.8 | 637.1 | 637.3 | 2888874 |
| 7 | 62.8 | 512.6 | 512.8 | 2937258 |
| 8 | 62.8 | 420.3 | 420.5 | 2984539 |
| 9 | 62.8 | 402.9 | 403.1 | 3031988 |
| 10 | 62.7 | 371.2 | 371.3 | 3078842 |
| 11 | 62.7 | 369.7 | 369.9 | 3127027 |
| 12 | 62.7 | 352.6 | 352.8 | 3174332 |
| 13 | 62.7 | 326.4 | 326.6 | 3221604 |
| 14 | 62.7 | 332.3 | 332.5 | 3265880 |
| 15 | 62.7 | 327.0 | 327.2 | 3310124 |
| 16 | 62.7 | 250.6 | 250.8 | 3353412 |
| 17 | 62.7 | 282.9 | 283.1 | 3396651 |
| 18 | 62.7 | 282.9 | 283.0 | 3442657 |
| 19 | 62.7 | 289.1 | 289.3 | 3488665 |
| 20 | 62.7 | 288.9 | 289.1 | 3533325 |
| 21 | 62.7 | 281.4 | 281.5 | 3578050 |
| 22 | 62.7 | 281.6 | 281.8 | 3622244 |
| 23 | 62.7 | 287.4 | 287.6 | 3666545 |
| 24 | 62.7 | 287.7 | 287.8 | 3709771 |
| 25 | 62.7 | 287.4 | 287.6 | 3753033 |
| 26 | 62.7 | 282.3 | 282.5 | 3799023 |
| 27 | 62.7 | 287.5 | 287.7 | 3845078 |
| 28 | 62.7 | 287.6 | 287.8 | 3889571 |
| 29 | 62.7 | 287.4 | 287.6 | 3934332 |
| 30 | 62.7 | 282.3 | 282.4 | 3980999 |
| 31 | 62.7 | 282.3 | 282.5 | 4027627 |

BCT: Time when best cover was initially found.
ST: Search time.
TT: Total program execution time.
EN: Number of expanded nodes
Time is in seconds.

Table D.6. Fine Grain Parallel SCP Statistics

| Searchers | 100.100.28.U | | | | 100.100.27.U | | | |
|---|---|---|---|---|---|---|---|---|
| | BCT | ST | TT | EN | BCT | ST | TT | EN |
| 1 | 148.2 | 6622.7 | 6622.9 | 3687497 | 5287.1 | 10527.2 | 10527.3 | 5935841 |
| 2 | 69.4 | 3468.2 | 3468.4 | 3682229 | 2644.5 | 5286.0 | 5286.3 | 5936310 |
| 3 | 56.0 | 2336.0 | 2336.2 | 3698337 | 1746.5 | 3678.9 | 3679.1 | 5909099 |
| 4 | 31.2 | 1876.3 | 1876.6 | 3674434 | 1321.1 | 2683.6 | 2683.9 | 5933016 |
| 5 | 27.2 | 1470.7 | 1471.0 | 3680854 | 926.0 | 2214.1 | 2214.3 | 5572711 |
| 6 | 24.2 | 1166.1 | 1166.3 | 3685586 | 883.7 | 1873.1 | 1873.3 | 5941782 |
| 7 | 23.0 | 1122.3 | 1122.5 | 3694223 | 531.7 | 1512.5 | 1512.7 | 5079816 |
| 8 | 21.1 | 1082.4 | 1082.6 | 3698111 | 495.5 | 1190.2 | 1190.4 | 5207998 |
| 9 | 20.1 | 798.4 | 798.7 | 3704545 | 487.1 | 1168.2 | 1168.4 | 5440144 |
| 10 | 19.1 | 776.6 | 776.9 | 3709939 | 464.4 | 1132.8 | 1133.0 | 5576679 |
| 11 | 18.1 | 753.7 | 753.9 | 3714594 | 457.0 | 1113.8 | 1114.1 | 5785604 |
| 12 | 17.1 | 736.7 | 737.0 | 3717880 | 445.1 | 1087.9 | 1088.1 | 5960363 |
| 13 | 17.1 | 734.0 | 734.3 | 3727005 | 101.7 | 758.2 | 758.4 | 3747866 |
| 14 | 17.0 | 717.4 | 717.7 | 3736301 | 101.4 | 757.0 | 757.2 | 3801246 |
| 15 | 16.1 | 704.6 | 704.9 | 3737579 | 99.8 | 755.0 | 755.3 | 3843267 |
| 16 | 16.1 | 703.1 | 703.4 | 3745773 | 81.7 | 451.9 | 452.2 | 3738898 |
| 17 | 16.0 | 442.8 | 443.0 | 3754164 | 80.3 | 452.7 | 452.9 | 3771077 |
| 18 | 16.0 | 445.2 | 445.5 | 3762641 | 79.9 | 451.5 | 451.8 | 3811261 |
| 19 | 15.2 | 428.6 | 428.8 | 3762019 | 66.2 | 438.5 | 438.8 | 3708021 |
| 20 | 14.3 | 427.1 | 427.4 | 3759498 | 66.0 | 438.0 | 438.3 | 3742168 |
| 21 | 14.2 | 426.2 | 426.4 | 3766641 | 66.0 | 438.0 | 438.3 | 3777661 |
| 22 | 13.3 | 425.3 | 425.5 | 3762989 | 66.0 | 437.5 | 437.8 | 3813236 |
| 23 | 13.3 | 425.9 | 426.2 | 3770066 | 65.8 | 437.1 | 437.3 | 3847844 |
| 24 | 13.3 | 425.8 | 426.0 | 3776597 | 61.5 | 432.4 | 432.6 | 3824981 |
| 25 | 13.3 | 425.4 | 425.7 | 3783547 | 61.4 | 432.0 | 432.3 | 3857064 |
| 26 | 13.2 | 425.0 | 425.3 | 3790043 | 61.4 | 432.2 | 432.5 | 3890799 |
| 27 | 13.2 | 424.5 | 424.8 | 3797040 | 61.4 | 420.9 | 421.2 | 3923863 |
| 28 | 13.2 | 425.1 | 425.4 | 3804077 | 61.3 | 420.8 | 421.1 | 3956600 |
| 29 | 13.2 | 425.1 | 425.4 | 3811031 | 61.2 | 420.8 | 421.0 | 3987824 |
| 30 | 12.4 | 424.4 | 424.7 | 3802772 | 60.5 | 416.4 | 416.7 | 4009502 |
| 31 | 12.4 | 424.4 | 424.6 | 3809159 | 60.3 | 415.9 | 416.2 | 4040440 |

BCT: Time when best cover was initially found.
ST: Search time.
TT: Total program execution time.
EN: Number of expanded nodes
Time is in seconds.

Table D.6 (Continued). Fine Grain Parallel SCP Statistics

| Searchers | 100.100.26.U | | | | 75.125.25.U | | | |
|---|---|---|---|---|---|---|---|---|
| | BCT | ST | TT | EN | BCT | ST | TT | EN |
| 1 | 1978.9 | 7355.6 | 7355.8 | 4199069 | 9512.1 | 9522.9 | 9523.1 | 6939581 |
| 2 | 985.9 | 3706.3 | 3706.5 | 4194865 | 4618.8 | 4629.6 | 4629.9 | 6707162 |
| 3 | 659.8 | 2650.3 | 2650.5 | 4198520 | 3221.4 | 3232.2 | 3232.4 | 6922129 |
| 4 | 485.9 | 1847.3 | 1847.5 | 4178267 | 2248.2 | 2259.0 | 2259.3 | 6534780 |
| 5 | 386.5 | 1682.9 | 1683.1 | 4170367 | 1761.6 | 1772.4 | 1772.7 | 6379487 |
| 6 | 318.4 | 1326.4 | 1326.7 | 4157115 | 1661.7 | 1672.5 | 1672.7 | 6976654 |
| 7 | 263.7 | 1239.4 | 1239.6 | 4119267 | 1308.3 | 1319.1 | 1319.3 | 6527171 |
| 8 | 244.9 | 964.5 | 964.7 | 4186511 | 1156.1 | 1167.0 | 1167.2 | 6317484 |
| 9 | 214.8 | 913.2 | 913.4 | 4167859 | 1088.8 | 1099.6 | 1099.9 | 6603089 |
| 10 | 181.7 | 863.9 | 864.2 | 4100894 | 743.6 | 754.4 | 754.6 | 5357216 |
| 11 | 176.9 | 844.8 | 845.9 | 4178387 | 736.7 | 747.4 | 747.7 | 5640400 |
| 12 | 159.9 | 825.3 | 825.6 | 4159656 | 701.0 | 711.8 | 712.1 | 5728083 |
| 13 | 142.8 | 794.3 | 794.6 | 4120399 | 670.3 | 681.1 | 681.3 | 5864526 |
| 14 | 126.0 | 763.7 | 763.9 | 4071034 | 641.6 | 652.4 | 652.6 | 5902102 |
| 15 | 112.9 | 749.7 | 750.0 | 4026857 | 636.1 | 646.9 | 647.2 | 6105426 |
| 16 | 109.0 | 486.9 | 487.1 | 4056131 | 633.2 | 644.0 | 644.3 | 6352589 |
| 17 | 108.2 | 485.7 | 486.0 | 4113402 | 613.4 | 624.2 | 624.5 | 6364104 |
| 18 | 95.9 | 472.6 | 472.8 | 4042812 | 612.3 | 623.1 | 623.4 | 6595488 |
| 19 | 91.9 | 468.2 | 468.4 | 4054668 | 268.5 | 279.3 | 279.6 | 3850921 |
| 20 | 91.1 | 467.3 | 467.6 | 4100710 | 255.8 | 267.2 | 267.5 | 3755432 |
| 21 | 91.1 | 466.4 | 466.7 | 4155120 | 255.2 | 266.6 | 266.9 | 3834753 |
| 22 | 91.1 | 453.2 | 453.5 | 4209410 | 251.0 | 262.5 | 262.8 | 3858573 |
| 23 | 74.1 | 434.1 | 434.3 | 4025482 | 233.8 | 245.3 | 245.5 | 3643287 |
| 24 | 74.1 | 434.0 | 434.2 | 4069320 | 233.3 | 244.8 | 245.1 | 3705944 |
| 25 | 74.1 | 434.3 | 434.6 | 4113191 | 233.3 | 244.8 | 245.1 | 3774625 |
| 26 | 74.0 | 434.6 | 434.8 | 4156950 | 233.0 | 244.5 | 244.7 | 3836593 |
| 27 | 73.2 | 432.7 | 433.0 | 4188464 | 231.5 | 243.1 | 243.3 | 3879565 |
| 28 | 57.3 | 418.1 | 418.4 | 3976902 | 230.5 | 242.0 | 242.3 | 3926341 |
| 29 | 57.3 | 418.0 | 418.3 | 4011165 | 229.2 | 240.7 | 241.0 | 3966820 |
| 30 | 45.0 | 405.7 | 405.9 | 3820110 | 229.3 | 240.9 | 241.1 | 4034643 |
| 31 | 41.0 | 401.1 | 401.3 | 3772995 | 229.4 | 241.0 | 241.1 | 4100518 |

BCT: Time when best cover was initially found.
ST: Search time.
TT: Total program execution time.
EN: Number of expanded nodes
Time is in seconds.

Table D.6 (Continued). Fine Grain Parallel SCP Statistics

| Searchers | 70.70.08.U | | | |
|---|---|---|---|---|
| | BCT | ST | TT | EN |
| 1 | 1870.5 | 4130.4 | 4130.5 | 3877454 |
| 2 | 687.6 | 2232.5 | 2232.6 | 3548905 |
| 3 | 325.0 | 1777.7 | 1777.8 | 3282476 |
| 4 | 188.6 | 1606.3 | 1606.4 | 3136370 |
| 5 | 100.2 | 1509.8 | 1509.9 | 2966949 |
| 6 | 96.6 | 1478.6 | 1478.7 | 3021477 |
| 7 | 72.5 | 1465.5 | 1465.7 | 2954274 |
| 8 | 70.7 | 1455.9 | 1456.1 | 2977564 |
| 9 | 63.2 | 1446.7 | 1446.9 | 2961082 |
| 10 | 61.9 | 1443.1 | 1443.3 | 2967481 |
| 11 | 59.4 | 1441.1 | 1441.2 | 2966104 |
| 12 | 59.2 | 1440.2 | 1440.4 | 2973345 |
| 13 | 58.9 | 1444.8 | 1445.0 | 2976463 |
| 14 | 58.8 | 1444.5 | 1444.6 | 2981229 |
| 15 | 58.4 | 1444.2 | 1444.4 | 2982121 |
| 16 | 58.4 | 1444.2 | 1444.4 | 2985078 |
| 17 | 58.5 | 1443.2 | 1443.3 | 2987752 |
| 18 | 58.2 | 1439.5 | 1439.7 | 2988756 |
| 19 | 58.2 | 1439.1 | 1439.3 | 2990224 |
| 20 | 58.4 | 1440.2 | 1440.3 | 2990322 |
| 21 | 58.4 | 1444.2 | 1444.3 | 2990671 |
| 22 | 58.3 | 1442.6 | 1442.8 | 2990173 |
| 23 | 58.4 | 1442.8 | 1442.9 | 2990728 |
| 24 | 58.2 | 1443.9 | 1444.1 | 2990112 |
| 25 | 58.2 | 1444.0 | 1444.1 | 2990104 |
| 26 | 58.3 | 1439.1 | 1439.3 | 2990287 |
| 27 | 58.3 | 1439.6 | 1439.8 | 2990097 |
| 28 | 58.4 | 1439.3 | 1439.4 | 2990611 |
| 29 | 58.3 | 1442.9 | 1443.1 | 2990067 |
| 30 | 58.3 | 1439.1 | 1439.3 | 2990169 |
| 31 | 58.3 | 1439.1 | 1439.3 | 2990190 |

BCT: Time when best cover was initially found.
ST: Search time.
TT: Total program execution time.
EN: Number of expanded nodes
Time is in seconds.

Table D.7. Dynamic Load Balanced Parallel SCP Statistics

| Searchers | 100.100.28.U | | | | 100.100.27.U | | | |
|---|---|---|---|---|---|---|---|---|
| | BCT | ST | TT | EN | BCT | ST | TT | EN |
| 1 | 207.2 | 9254.5 | 9254.7 | 3687497 | 7388.3 | 14712.9 | 14713.1 | 5935841 |
| 2 | 97.0 | 4619.1 | 4619.3 | 3682307 | 3692.6 | 7352.6 | 7352.8 | 5936581 |
| 3 | 78.2 | 3095.8 | 3096.0 | 3698483 | 2441.6 | 4886.1 | 4886.3 | 5911116 |
| 4 | 43.5 | 2346.1 | 2346.3 | 3731342 | 1848.7 | 3684.2 | 3684.5 | 5934329 |
| 5 | 37.9 | 1870.6 | 1870.9 | 3719722 | 1296.4 | 2818.4 | 2818.7 | 5657347 |
| 6 | 33.7 | 1545.0 | 1545.3 | 3685644 | 1236.7 | 2471.5 | 2471.8 | 5963108 |
| 7 | 32.0 | 1356.8 | 1357.0 | 3771108 | 743.7 | 1854.5 | 1854.8 | 5201384 |
| 8 | 29.3 | 1251.3 | 1251.6 | 3952961 | 693.4 | 1633.0 | 1633.3 | 5237539 |
| 9 | 28.0 | 1050.5 | 1050.8 | 3738090 | 680.9 | 1507.1 | 1507.3 | 5439645 |
| 10 | 26.6 | 971.2 | 971.5 | 3841882 | 647.4 | 1389.8 | 1390.0 | 5569151 |
| 11 | 25.2 | 895.4 | 895.7 | 3893483 | 639.1 | 1354.3 | 1354.6 | 5962094 |
| 12 | 23.9 | 824.3 | 824.6 | 3904916 | 622.4 | 1271.5 | 1271.7 | 6107964 |
| 13 | 23.8 | 782.8 | 783.1 | 3999442 | 142.4 | 793.4 | 793.6 | 4036731 |
| 14 | 23.7 | 741.0 | 741.2 | 4061461 | 141.5 | 773.4 | 773.6 | 4201892 |
| 15 | 22.4 | 711.7 | 711.9 | 4155221 | 139.4 | 768.1 | 768.4 | 4394147 |
| 16 | 22.3 | 711.1 | 711.4 | 4332438 | 114.1 | 604.7 | 605.0 | 3798807 |
| 17 | 22.2 | 581.1 | 581.3 | 3895273 | 82.3 | 421.1 | 421.3 | 3824427 |
| 18 | 22.2 | 557.1 | 557.4 | 3931887 | 81.9 | 405.4 | 405.6 | 3818457 |
| 19 | 21.1 | 516.9 | 517.2 | 3814800 | 67.9 | 386.5 | 386.8 | 3863002 |
| 20 | 19.8 | 517.2 | 517.5 | 3992884 | 67.7 | 384.6 | 384.9 | 4019697 |
| 21 | 19.8 | 506.3 | 506.6 | 4095028 | 67.6 | 380.0 | 380.3 | 4173330 |
| 22 | 18.5 | 509.6 | 509.8 | 4324991 | 67.6 | 377.4 | 377.7 | 4347725 |
| 23 | 18.4 | 504.8 | 505.1 | 4446322 | 67.4 | 381.1 | 381.4 | 4546100 |
| 24 | 18.4 | 497.3 | 497.6 | 4569803 | 63.0 | 376.2 | 376.5 | 4674789 |
| 25 | 18.4 | 489.5 | 489.7 | 4676441 | 62.9 | 375.6 | 375.9 | 4876086 |
| 26 | 18.3 | 498.1 | 498.4 | 4979991 | 62.9 | 365.3 | 365.6 | 4917570 |
| 27 | 18.3 | 503.4 | 503.6 | 5154768 | 62.9 | 364.5 | 364.8 | 5107966 |
| 28 | 18.3 | 473.1 | 473.4 | 5040636 | 62.8 | 370.6 | 370.9 | 5348620 |
| 29 | 18.3 | 467.4 | 467.7 | 5125823 | 62.7 | 368.5 | 368.7 | 5441404 |
| 30 | 17.1 | 457.2 | 457.4 | 5152413 | 61.9 | 358.5 | 358.8 | 5531284 |
| 31 | 17.1 | 460.2 | 460.4 | 5377656 | 61.8 | 369.1 | 369.4 | 5815763 |

BCT: Time when best cover was initially found.
ST: Search time.
TT: Total program execution time.
EN: Number of expanded nodes
Time is in seconds.

Table D.7 (Continued). Dynamic Load Balanced Parallel SCP Statistics

| Searchers | 100.100.26.U | | | | 75.125.25.U | | | |
|---|---|---|---|---|---|---|---|---|
| | BCT | ST | TT | EN | BCT | ST | TT | EN |
| 1 | 2029.0 | 7537.8 | 7538.0 | 4199069 | 9783.7 | 9794.7 | 9795.0 | 6939581 |
| 2 | 1010.7 | 3766.0 | 3766.2 | 4194865 | 4730.5 | 4736.8 | 4737.1 | 6701992 |
| 3 | 676.5 | 2515.7 | 2516.0 | 4198599 | 3249.6 | 3254.6 | 3254.9 | 6917840 |
| 4 | 498.2 | 1878.3 | 1878.5 | 4179196 | 2304.4 | 2311.2 | 2311.5 | 6543207 |
| 5 | 396.2 | 1530.8 | 1531.1 | 4244902 | 1791.8 | 1800.9 | 1801.1 | 6375185 |
| 6 | 326.4 | 1249.3 | 1249.5 | 4158144 | 1633.1 | 1638.3 | 1638.6 | 6949448 |
| 7 | 270.3 | 1094.1 | 1094.3 | 4237436 | 1305.0 | 1313.4 | 1313.6 | 6509844 |
| 8 | 251.1 | 950.3 | 950.5 | 4216079 | 1109.4 | 1115.0 | 1115.3 | 6291918 |
| 9 | 220.2 | 843.9 | 844.1 | 4194949 | 1045.8 | 1051.2 | 1051.4 | 6656920 |
| 10 | 186.2 | 764.7 | 765.0 | 4220185 | 755.8 | 765.3 | 765.6 | 5387560 |
| 11 | 181.3 | 717.5 | 717.8 | 4348090 | 698.0 | 707.4 | 707.7 | 5480261 |
| 12 | 163.9 | 689.9 | 690.2 | 4537361 | 707.6 | 716.0 | 716.2 | 6024297 |
| 13 | 146.4 | 639.9 | 640.1 | 4535682 | 646.3 | 653.6 | 653.8 | 5967879 |
| 14 | 129.2 | 580.6 | 580.9 | 4410621 | 579.6 | 587.6 | 588.0 | 5771671 |
| 15 | 115.8 | 556.6 | 556.8 | 4453494 | 598.7 | 605.1 | 605.3 | 6349913 |
| 16 | 111.7 | 478.4 | 478.6 | 4204050 | 508.6 | 582.4 | 582.7 | 6491078 |
| 17 | 110.9 | 454.7 | 455.0 | 4183504 | 554.0 | 560.8 | 561.1 | 6634078 |
| 18 | 98.3 | 418.6 | 418.9 | 4116641 | 535.2 | 540.0 | 540.3 | 6776411 |
| 19 | 94.2 | 424.9 | 425.1 | 4320563 | 276.3 | 287.3 | 287.3 | 3851439 |
| 20 | 93.4 | 409.4 | 409.6 | 4449380 | 263.6 | 274.5 | 274.8 | 3851907 |
| 21 | 93.4 | 406.2 | 406.4 | 4551921 | 262.1 | 272.1 | 272.4 | 4013897 |
| 22 | 93.4 | 406.0 | 406.2 | 4812692 | 258.4 | 268.9 | 269.2 | 4140161 |
| 23 | 75.9 | 373.6 | 373.9 | 4563806 | 240.3 | 250.9 | 251.2 | 4015712 |
| 24 | 75.9 | 370.8 | 371.1 | 4709982 | 199.3 | 211.4 | 211.7 | 3531648 |
| 25 | 75.9 | 367.3 | 367.6 | 4867595 | 239.7 | 250.2 | 250.5 | 4328303 |
| 26 | 75.9 | 370.1 | 370.3 | 5071190 | 239.4 | 250.7 | 251.0 | 4491268 |
| 27 | 75.1 | 359.5 | 359.8 | 5165696 | 238.6 | 249.7 | 249.9 | 4634800 |
| 28 | 58.7 | 345.7 | 346.0 | 5027405 | 237.5 | 248.7 | 248.9 | 4800440 |
| 29 | 58.7 | 353.8 | 354.0 | 5336114 | 220.1 | 232.1 | 232.3 | 4634845 |
| 30 | 46.1 | 340.9 | 341.1 | 5240828 | 236.1 | 246.7 | 247.0 | 5087735 |
| 31 | 42.1 | 344.6 | 344.8 | 5467881 | 236.0 | 246.6 | 246.9 | 5215345 |

BCT: Time when best cover was initially found.
ST: Search time.
TT: Total program execution time.
EN: Number of expanded nodes
Time is in seconds.

Table D.7 (Continued). Dynamic Load Balanced Parallel SCP Statistics

| Searchers | 70.70.08.U | | | |
|---|---|---|---|---|
| | BCT | ST | TT | EN |
| 1 | 1909.5 | 4214.1 | 4214.3 | 3877454 |
| 2 | 702.1 | 1932.7 | 1932.8 | 3549111 |
| 3 | 331.7 | 1195.2 | 1195.4 | 3282334 |
| 4 | 192.8 | 867.3 | 867.5 | 3169786 |
| 5 | 102.5 | 664.9 | 665.1 | 3029919 |
| 6 | 98.8 | 565.2 | 565.4 | 3086566 |
| 7 | 74.1 | 482.6 | 482.7 | 3067194 |
| 8 | 72.3 | 429.0 | 429.2 | 3116936 |
| 9 | 64.8 | 403.0 | 403.2 | 3280278 |
| 10 | 63.4 | 350.5 | 350.7 | 3150803 |
| 11 | 60.6 | 329.4 | 329.6 | 3258935 |
| 12 | 60.6 | 318.0 | 318.2 | 3412913 |
| 13 | 60.0 | 296.6 | 296.8 | 3458709 |
| 14 | 60.0 | 286.4 | 286.6 | 3582617 |
| 15 | 60.0 | 295.0 | 295.1 | 3944612 |
| 16 | 59.8 | 274.9 | 275.1 | 3941925 |
| 17 | 59.9 | 252.2 | 252.4 | 3802190 |
| 18 | 59.6 | 279.9 | 280.0 | 4429549 |
| 19 | 59.6 | 251.1 | 251.3 | 4103877 |
| 20 | 59.7 | 246.2 | 246.4 | 4252316 |
| 21 | 59.7 | 265.7 | 265.9 | 4799182 |
| 22 | 59.8 | 285.8 | 286.0 | 5374693 |
| 23 | 59.7 | 265.0 | 265.1 | 5041567 |
| 24 | 59.8 | 254.5 | 254.7 | 5101067 |
| 25 | 59.8 | 285.9 | 286.1 | 5937413 |
| 26 | 59.7 | 256.6 | 256.8 | 5558216 |
| 27 | 59.7 | 267.1 | 267.3 | 5953949 |
| 28 | 59.7 | 228.9 | 229.1 | 5235085 |
| 29 | 59.7 | 238.2 | 238.4 | 5591061 |
| 30 | 59.7 | 254.3 | 254.5 | 6408799 |
| 31 | 59.7 | 244.8 | 244.9 | 6199314 |

BCT: Time when best cover was initially found.
ST: Search time.
TT: Total program execution time.
EN: Number of expanded nodes
Time is in seconds.

Table D.8. Tabulated Speedup Data

| Searchers | 100.100.28.U | | | 100.100.27.U | | |
|---|---|---|---|---|---|---|
| | Coarse Grain | Fine Grain | Load Balance | Coarse Grain | Fine Grain | Load Balance |
| 1 | 0.4 | 1.0 | 1.0 | 0.5 | 1.0 | 0.7 |
| 2 | 0.7 | 2.0 | 1.5 | 1.2 | 2.1 | 1.5 |
| 3 | 1.0 | 3.0 | 2.2 | 1.7 | 3.0 | 2.2 |
| 4 | 1.4 | 3.7 | 3.0 | 2.1 | 4.1 | 3.0 |
| 5 | 1.8 | 4.7 | 3.7 | 2.4 | 4.9 | 3.9 |
| 6 | 2.0 | 5.9 | 4.5 | 3.0 | 5.8 | 4.4 |
| 7 | 2.2 | 6.2 | 5.1 | 3.0 | 7.2 | 5.9 |
| 8 | 2.5 | 6.4 | 5.5 | 3.3 | 9.2 | 6.7 |
| 9 | 3.0 | 8.7 | 6.6 | 4.5 | 9.4 | 7.3 |
| 10 | 3.1 | 8.9 | 7.1 | 4.5 | 9.7 | 7.9 |
| 11 | 3.5 | 9.2 | 7.7 | 4.5 | 9.8 | 8.1 |
| 12 | 3.1 | 9.4 | 8.4 | 6.1 | 10.1 | 8.6 |
| 13 | 3.6 | 9.4 | 8.8 | 6.1 | 14.4 | 13.8 |
| 14 | 4.4 | 9.7 | 9.3 | 6.1 | 14.5 | 14.2 |
| 15 | 4.4 | 9.8 | 9.7 | 6.2 | 14.5 | 14.2 |
| 16 | 4.4 | 15.6 | 9.7 | 6.3 | 24.2 | 18.1 |
| 17 | 4.6 | 15.6 | 11.9 | 7.6 | 24.2 | 26.0 |
| 18 | 5.8 | 16.2 | 12.4 | 7.6 | 24.2 | 27.0 |
| 19 | 5.8 | 16.2 | 13.4 | 7.6 | 25.0 | 28.3 |
| 20 | 5.8 | 16.2 | 13.4 | 7.6 | 25.0 | 28.5 |
| 21 | 5.8 | 16.3 | 13.7 | 7.8 | 25.0 | 28.8 |
| 22 | 6.0 | 16.3 | 13.6 | 7.8 | 25.0 | 29.0 |
| 23 | 5.9 | 16.3 | 13.7 | 7.8 | 25.0 | 28.7 |
| 24 | 5.9 | 16.3 | 13.9 | 7.8 | 25.3 | 29.1 |
| 25 | 6.2 | 16.3 | 14.1 | 7.9 | 25.3 | 29.1 |
| 26 | 6.2 | 16.3 | 13.9 | 10.4 | 25.3 | 30.0 |
| 27 | 6.2 | 16.3 | 13.8 | 10.3 | 26.0 | 30.0 |
| 28 | 8.7 | 16.3 | 14.6 | 10.3 | 26.0 | 29.5 |
| 29 | 6.2 | 16.3 | 14.8 | 10.3 | 26.0 | 29.7 |
| 30 | 6.2 | 16.3 | 15.1 | 10.3 | 26.3 | 30.5 |
| 31 | 6.2 | 16.3 | 15.0 | 10.3 | 26.3 | 29.6 |

Table D.8 (Continued). Tabulated Speedup Data

| | 100.100.26.U | | | 75.125.25.U | | |
|---|---|---|---|---|---|---|
| Searchers | Coarse Grain | Fine Grain | Load Balance | Coarse Grain | Fine Grain | Load Balance |
| 1 | 0.7 | 0.8 | 0.7 | 1.2 | 1.0 | 1.0 |
| 2 | 1.1 | 1.5 | 1.5 | 2.6 | 2.0 | 2.0 |
| 3 | 3.2 | 2.1 | 2.2 | 3.4 | 2.9 | 2.9 |
| 4 | 4.2 | 3.0 | 3.0 | 4.8 | 4.1 | 4.1 |
| 5 | 4.5 | 3.3 | 3.6 | 6.0 | 5.3 | 5.2 |
| 6 | 6.1 | 4.2 | 4.5 | 9.5 | 5.6 | 5.7 |
| 7 | 6.2 | 4.5 | 5.1 | 14.3 | 7.1 | 7.1 |
| 8 | 6.4 | 5.8 | 5.9 | 14.8 | 8.0 | 8.4 |
| 9 | 6.7 | 6.1 | 6.6 | 16.0 | 8.5 | 8.9 |
| 10 | 7.4 | 6.5 | 7.3 | 14.9 | 12.4 | 12.2 |
| 11 | 7.7 | 6.6 | 7.8 | 15.4 | 12.5 | 13.2 |
| 12 | 12.6 | 6.8 | 8.1 | 15.3 | 13.2 | 13.1 |
| 13 | 12.6 | 7.0 | 8.7 | 16.5 | 13.7 | 14.3 |
| 14 | 13.0 | 7.3 | 9.6 | 59.8 | 14.4 | 16.0 |
| 15 | 13.1 | 7.4 | 10.0 | 59.8 | 14.5 | 15.5 |
| 16 | 13.6 | 11.5 | 11.7 | 60.0 | 14.5 | 16.1 |
| 17 | 13.6 | 11.5 | 12.3 | 60.2 | 15.0 | 16.7 |
| 18 | 13.6 | 11.8 | 13.3 | 60.2 | 15.0 | 17.3 |
| 19 | 14.1 | 11.9 | 13.1 | 60.2 | 33.5 | 32.6 |
| 20 | 14.1 | 11.9 | 13.6 | 60.6 | 35.0 | 34.1 |
| 21 | 14.1 | 12.0 | 13.7 | 60.6 | 35.1 | 34.4 |
| 22 | 14.1 | 12.3 | 13.7 | 60.4 | 35.7 | 34.8 |
| 23 | 14.2 | 12.9 | 14.9 | 60.8 | 38.2 | 37.3 |
| 24 | 14.7 | 12.9 | 15.0 | 60.8 | 38.3 | 44.3 |
| 25 | 14.7 | 12.8 | 15.2 | 60.8 | 38.3 | 37.4 |
| 26 | 14.7 | 12.8 | 15.1 | 60.8 | 38.3 | 37.3 |
| 27 | 14.7 | 12.9 | 15.5 | 61.0 | 38.5 | 37.5 |
| 28 | 14.7 | 12.3 | 16.1 | 61.1 | 38.7 | 37.6 |
| 29 | 14.7 | 12.3 | 15.8 | 61.1 | 38.9 | 40.3 |
| 30 | 14.7 | 13.3 | 16.4 | 61.0 | 38.9 | 38.0 |
| 31 | 14.7 | 13.8 | 16.2 | 61.0 | 38.9 | 38.0 |

Table D.8 (Continued). Tabulated Speedup Data

| Searchers | 70.70.08.U | | |
|---|---|---|---|
| | Coarse Grain | Fine Grain | Load Balance |
| 1 | 1.2 | 1.1 | 1.1 |
| 2 | 2.7 | 2.0 | 2.3 |
| 3 | 3.8 | 2.5 | 3.7 |
| 4 | 5.1 | 2.8 | 5.1 |
| 5 | 5.7 | 2.9 | 6.7 |
| 6 | 7.0 | 3.0 | 7.9 |
| 7 | 8.7 | 3.0 | 9.2 |
| 8 | 10.6 | 3.1 | 10.4 |
| 9 | 11.0 | 3.1 | 11.0 |
| 10 | 12.0 | 3.1 | 12.7 |
| 11 | 12.0 | 3.1 | 13.5 |
| 12 | 12.6 | 3.1 | 14.0 |
| 13 | 13.6 | 3.1 | 15.0 |
| 14 | 13.4 | 3.1 | 15.5 |
| 15 | 13.6 | 3.1 | 15.1 |
| 16 | 17.3 | 3.1 | 16.2 |
| 17 | 15.7 | 3.1 | 17.6 |
| 18 | 15.7 | 3.1 | 15.9 |
| 19 | 15.4 | 3.1 | 17.7 |
| 20 | 15.4 | 3.1 | 18.0 |
| 21 | 15.8 | 3.1 | 16.7 |
| 22 | 15.8 | 3.1 | 15.5 |
| 23 | 15.5 | 3.1 | 16.8 |
| 24 | 15.4 | 3.1 | 17.5 |
| 25 | 15.5 | 3.1 | 15.5 |
| 26 | 15.7 | 3.1 | 17.3 |
| 27 | 15.5 | 3.1 | 16.6 |
| 28 | 15.4 | 3.1 | 19.4 |
| 29 | 15.5 | 3.1 | 18.7 |
| 30 | 15.7 | 3.1 | 17.5 |
| 31 | 15.7 | 3.1 | 18.2 |

# Appendix E. *NP-Complete Problems*

## *E.1 Introduction*

Many of the real-world problems mentioned in Chapter I are NP-complete and may be solved using optimal search techniques developed to solve the following NP-complete problems (4, 25):

Assignment Problem
Hamiltonian Circuit Problem
Traveling Salesman Problem
0/1 Knapsack Problem
Set Covering Problem (SCP)

The purpose of this appendix is to describe the above NP-complete problems with the proofs of their NP-completeness left to Aho (4). The first four problems are briefly described in Sections E.2, E.3, E.4, and E.5. Since this research explores the parallelization of the SCP, a detailed explanation of the SCP and its component parts is presented in Section E.6.

## *E.2 Assignment Problem*

The general assignment problem is the problem of assigning $w$ resources to $t$ tasks subject to a set of constraint(s). An effective implementation of this problem finds an optimal assignment, but depending on the values of $w$ and $t$, this problem may not be NP-complete. For example, $w^3$ time is required to solve an assignment problem where $w = t$. If, however, $w < t$, the problem is NP-complete requiring $O(w^t)$ time to compute an optimal solution in the worst case (19). The notation, O( ), is used to indicate "order-of". This order-of refers to an upper, lower, or exact-bound on the number of calculations required to solve the problem. In other words, it refers to the *efficiency* of the problem's algorithm. In many cases, *efficiency* is used to indicated time and space requirements. In this document, roughly all order-of or complexity analysis assumes the worst-case or upper-bound on the number of calculations and the time required to compute those calculations.

The order-of analysis for space requirements is only included where relevant. Figure E.1 shows a complete assignment tree for $t = 3$ and $w = 2$. One assignment from this tree might be T1 assigned to W1, T2 assigned to W1, and T3 assigned to W2 for a total cost of C2. Notice that every combination of $w$ and $t$ is represented. The optimal assignment is that combination which results in the lowest cost $c$.

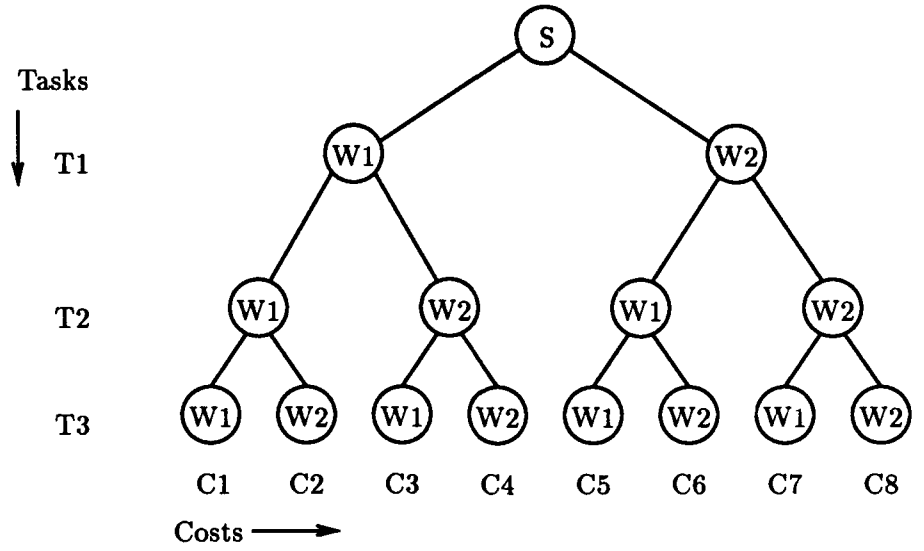



Figure E.1. Tree Representing $w$ Weapons Assigned to $t$ Targets

### *E.3 Hamiltonian Circuit Problem*

Given a directed graph, Figure E.2, composed of vertices and arcs, a circuit in this graph is a directed path with the same starting and ending vertex. The Hamiltonian Circuit Problem is the problem of finding a circuit in the graph such that no vertex is used more than once (an elementary circuit) and the circuit encompasses all vertices (17:6). In Figure E.2, the Hamiltonian circuits are $\{a \rightarrow b \rightarrow c \rightarrow d \rightarrow a\}$, $\{a \rightarrow b \rightarrow d \rightarrow c \rightarrow a\}$, and $\{a \rightarrow c \rightarrow b \rightarrow d \rightarrow a\}$.

### *E.4 Traveling Salesman Problem*

Now, let costs be associated with the arcs of the directed graph such that the cost of the circuit is the sum of the arcs comprising the circuit. The traveling salesman problem

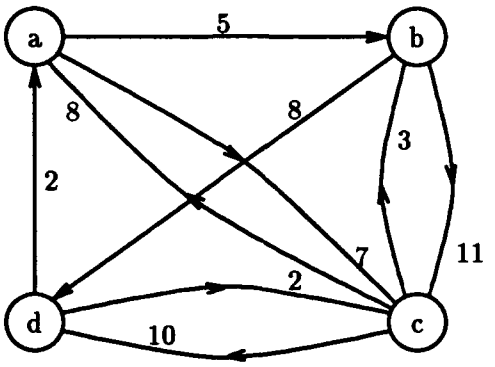



Figure E.2. Directed Graph

(TSP) is the problem of finding the minimum cost Hamiltonian circuit (25:231). In Figure E.2, the solution to the TSP is given by the Hamiltonian circuit $\{a \rightarrow c \rightarrow b \rightarrow d \rightarrow a\}$ with a cost of 20.

### *E.5 0/1 Knapsack Problem*

"Given $n$ positive weights $w_i$, $n$ positive profits $p_i$, and a positive number $M$ which is the knapsack capacity, this problem calls for choosing a subset of the weights such that

$$\sum_{1 \leq i \leq n} w_i x_i \leq M \quad \text{and} \quad \sum_{1 \leq i \leq n} p_i x_i \text{ is maximized}$$

The $x$'s constitute a zero-one valued vector" (29:350). In other words, if object $i$ is placed in the knapsack, a profit of $p_i$ is earned. The objective of the knapsack problem is to fill the knapsack such that the profit is maximized.

### *E.6 Set Covering Problem*

The set covering problem (SCP) is one of a large class of NP-complete problems (see Aho (4:392) for a proof of the SCP's NP-completeness). It was extensively studied in the late 1960's and early 1970's in connection with operational research problems and many articles are available from this time period covering search, reduction, and applications (18, 7, 52, 27, 43). The following sections present a syntactic definition of the SCP, three

example applications, the search technique to be used to find the optimal cover, four reduction techniques, a dominance test, and a lower bound test.

*E.6.1 Description* The SCP may be defined by the following definition (17:39):

> Given a set $R = \{r_1, r_2, \ldots, r_m\}$ and a family $\mathcal{L} = \{S_1, S_2, \ldots, S_N\}$ of sets such that $S_j \subset R$, any subfamily of $\mathcal{L} = \{S_{j1}, S_{j2}, \ldots, S_{jk}\}$ such that
>
> $$\bigcup_{i=1}^{k} S_{ji} = R \tag{E.1}$$
>
> is called a set covering of $R$.
>
> Given a 0–1 matrix,
>
> $$\begin{aligned} \text{Minimize:} \quad & z = \sum_{j=1}^{N} c_j \nu_j \\ \text{Subject to:} \quad & \sum_{j=1}^{N} t_{ij} \nu_j, i = 1, 2, 3, \ldots, m \end{aligned} \tag{E.2}$$

In a less syntactic definition, minimize the cost such that all the elements of $R$ are covered by at least one set from $\mathcal{L}$ (17:39). The objective of the SCP is to cover all the vertices in a graph with the minimum cost set of arcs or to find the set of arcs such that all vertices have at least one arc and the set of arcs have a cost lower than any other set of arcs which also cover the vertices. Another way to view the problem is to consider an 0–1 matrix such as the one shown in Figure 2.1 and repeated in this section. In this matrix, the objective is to cover all the rows (vertices) with some subset of the columns (arcs or sets) such that the total cost of the subset of columns is less than any other covering subset.

As in the previous sections on search techniques, Figure 2.1 is used to illustrate the concepts presented in this section. In Figure 2.1, the sets {0, 1, 2, 3, 4}, {3, 4, 5, 6}, and {0, 3, 4} each cover all the vertices with costs 27, 19, and 15 respectively. These sets are not all the covering sets, just a representative sample. The solution to this SCP is the set {0, 3, 4} which has a cost of 15. Any other combination of sets which cover the vertices have a cost greater than the cost of this set. Before proceeding with a discussion of the search technique, the next section presents three example applications of the SCP.

| | | Sets | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 |
| | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 |
| Vertices | 2 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| | 3 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 1 |
| | 4 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 |
| | 5 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 |
| | | 4 | 7 | 5 | 8 | 3 | 2 | 6 | 5 |
| | | Costs | | | | | | | |

Figure 2.1. 0–1 Matrix For a Set Covering Problem (17:54)

*E.6.2 SCP Applications* The SCP has application in a number of different fields. For example, airline and assembly line scheduling, design of computer systems, railroad-crew scheduling, and political districting are all types of problems which can be formulated as an SCP (17:591) (56:94) (7:1152). Furthermore, since the SCP is an NP-complete problem, it can be used to solve other NP-complete problems such as the assignment and graph coloring problems. The key to applying the SCP to any of these problems is to identify the items that must be covered by some subset of another list of items. Once the two lists of items are identified, they must be formulated as a 0–1 matrix with the items to be covered as the rows and the covering items as the columns. Additionally, the covering items must have some associated cost to identify their relative importance. The intent of these examples is not to argue that the SCP should be used to solve these types of problems. They merely illustrate the general applicability of the SCP to actual and NP-complete problems.

*E.6.2.1 Selection of Interpreters* Associate some semantics to the rows and columns of Figure 2.1 as shown in Figure E.3.

Let the rows represent a list of languages which are required to be interpreted and let the columns represent perspective interpreters. A '1' in any row/column means that the perspective interpreter can speak that language. The salary for each interpreter is represented at the bottom of the matrix. The objective then is to choose a subset of the interpreters such that all the languages are spoken by at least one person and that the

| | | Applicants | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | A | B | C | D | E | F | G | H |
| Languages | French | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 |
| | German | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 |
| | Chinese | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| | Spanish | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 1 |
| | Italian | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 |
| | English | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 |
| | | 4 | 7 | 5 | 8 | 3 | 2 | 6 | 5 |
| | | Costs | | | | | | | |

Figure E.3. 0–1 Matrix for the Selection of Interpreters

total salary of the interpreters is minimal.

It was previously stated that the cover for this matrix is {A, D, E} for a cost of 15. Based on this cover, interpreters A, D, and E can speak all the languages required to be spoken for the minimum salary (17:47,54).

*E.6.2.2 Assignment Problem* The assignment problem is also solvable as an SCP problem. Let $T$ targets be the items to be covered and let $W$ weapons be the covering items. List the targets as the rows and, for each weapon, construct columns corresponding to targets the weapon can attack. Assign a cost to each column based on the cost to assign that particular weapon to the list of targets. A low cost for a weapon to cover a list of targets corresponds to a high priority assignment. Figure E.4 is an example of the weapon assignment 0–1 matrix. The resulting assignment covers all targets with some combination of weapons. In this example, the matrix must be dynamic since targets and weapons are destroyed in real-time. Thus, the SCP must be executed after each iteration of weapon firing to constantly obtain new assignments.

*E.6.2.3 Graph Coloring Problem* The graph coloring problem is another NP-complete problem which is easily solvable as an SCP. The objective in a graph coloring problem is to color the vertices of a graph such that no two adjacent vertices are the same color. A good illustration is a map of the United States. The individual states of the map must be colored so that each state is identifiable. The states are represented as vertices of a

| | | Weapons | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | 1 | 2 | 3 | 4 | 3 | 2 | 4 | 1 | 2 | 1 | 4 | 4 | 3 | 2 | 1 |
| Targets | $T_1$ | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 |
| | $T_2$ | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| | $T_3$ | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 |
| | $T_4$ | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 0 |
| | $T_5$ | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 1 |
| | $T_6$ | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 1 |
| | | 1 | 5 | 8 | 4 | 2 | 9 | 6 | 8 | 3 | 1 | 1 | 2 | 4 | 3 | 4 |
| | | Costs | | | | | | | | | | | | | | |

Figure E.4. Assignment Problem 0–1 Matrix

graph and the borders between states are links between the vertices. Hence, any two states that share the same border are adjacent and must be different colors. The general process for solving a graph coloring problem with the SCP is to build an adjacency matrix of the items to be colored, compute the maximal independent sets (MISs) from the adjacency matrix, construct a 0–1 matrix containing the items as rows and the MISs as columns, and then cover the items with the MISs.

An adjacency matrix is a 0–1 matrix with the vertices of the graph represented as rows and columns. If a row vertex and a column vertex are adjacent (i.e., there is a path of length one between them) then a '1' is entered in the matrix. A '0' is entered in all row/column pairs which are not adjacent. A set is maximally independent if: (1) given a set of vertices, $\nu$, of a graph, no two vertices of $\nu$ are adjacent and (2) no other set of vertices, $\lambda$, contains $\nu$. A graph may contain numerous MISs and the cardinality of each MIS may be different. An algorithm for generating all MISs is given by Christofides (17:31–32).

Consider the undirected graph of Figure E.5. The adjacency matrix for this graph is represented in Figure E.6. The six vertices are placed on the rows and columns and a '1' is inserted in every row/column corresponding to adjacent vertices.

The maximal independent sets are generated and placed in a 0–1 matrix as in Figure E.7. The rows now represent the vertices in the graph and the columns are the maximal independent sets. Costs must be associated to the columns because the SCP finds a minimum cost cover. All the columns are equally important; therefore, each column gets a cost

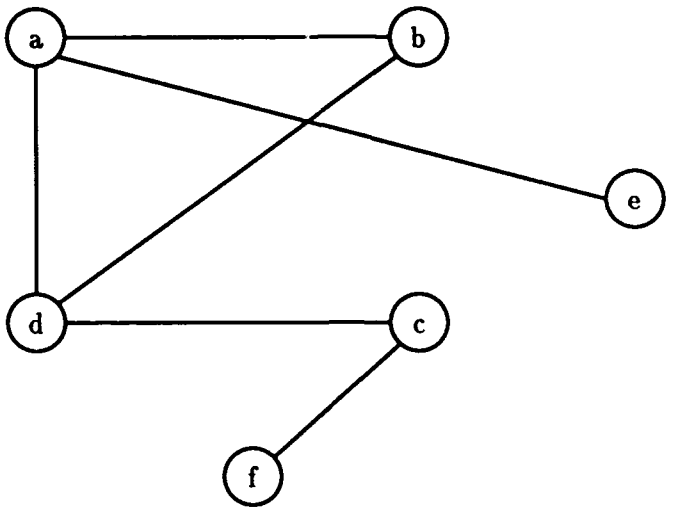



Figure E.5. Undirected Graph

| | a | b | c | d | e | f |
|---|---|---|---|---|---|---|
| a | 0 | 1 | 0 | 1 | 1 | 0 |
| b | 1 | 0 | 0 | 1 | 0 | 0 |
| c | 0 | 0 | 0 | 1 | 0 | 1 |
| d | 1 | 1 | 1 | 0 | 0 | 0 |
| e | 1 | 0 | 0 | 0 | 0 | 0 |
| f | 0 | 0 | 1 | 0 | 0 | 0 |

Figure E.6. Adjacency Matrix for Undirected Graph of Figure E.5

| | $S_1$ | $S_2$ | $S_3$ | $S_4$ | $S_5$ |
|---|---|---|---|---|---|
| a | 1 | 1 | 0 | 0 | 0 |
| b | 0 | 0 | 1 | 1 | 0 |
| c | 1 | 0 | 1 | 0 | 0 |
| d | 0 | 0 | 0 | 0 | 1 |
| e | 0 | 0 | 1 | 1 | 1 |
| f | 0 | 1 | 0 | 1 | 1 |
| | 1 | 1 | 1 | 1 | 1 |
| | Costs | | | | |

Figure E.7. Maximal Independent Sets 0–1 Matrix

of one. A solution to this coloring problem is now found by covering the rows with the minimum number of columns. One solution to this covering problem is the sets $\{S_1, S_4, S_5\}$ for a cost of three. This solution is interpreted to mean that three colors are required to color the graph and that the following sets of vertices can be colored the same color: {a,c}, {b,e,f}, and {d}. Notice that $S_4$ and $S_5$ both contain {e,f}. This means that {e,f} can be either color (17:68–69).

*E.6.2.4 Other NP-complete Problems* In general, any NP-complete problem can be solved using the SCP since all NP-complete problems are transformable to each other in polynomial time (4:373). However, many transformations may be required to massage the problem until it is solvable as an SCP (4:385). Hence, it is not necessarily efficient to solve other NP-complete problems in this manner. Even so, the data and control structures necessary to search the state space are similar.

*E.6.3 Search Methodology* The optimal solution to the SCP involves a search for the least (or greatest) cost set of columns which cover all rows. Since it is a search, all search techniques presented in Section 2.4 apply. The main stipulation or restriction on the SCP search is that the entire search space must be checked either explicitly or implicitly to ensure the optimal solution is obtained. Rather than use the simple depth-first or breadth-first search, a branch-and-bound search is more efficient for reasons already presented in Section 2.4.6. At each stage of the search, the selection function chooses a column based on a depth-first expansion. The branching procedure ensures that the column does not currently belong to the cover. If the column has already been added, the search backtracks and the next column is chosen. The selection and branching procedures iterate until a new column is added to the current cover. The elimination procedure has two functions: 1) it compares a complete cover against the previously retained cover and keeps the better, 2) it performs dominance testing and lower bound computation to eliminate those portions of the search tree which can not possibly lead to a better solution. The termination test tracks the progression of the search and terminates when all covers are checked. Note that it is not necessary to expand all covers explicitly since the elimination procedure may eliminate entire subbranches of the search tree. Therefore, the termination test must track

the algorithm's current position in the search tree.

*E.6.4 Reductions and Preconditioning* Before the branch-and-bound search is executed, reductions and preconditioning of the search space are possible. Four polynomial-time reduction methods which potentially reduce the input problem's dimensions are outlined (17:40):

1. Reduction #1: The input matrix is examined for a row which has no cover. If such a row exists, then the problem is unsolvable and the search is complete.
2. Reduction #2: If there exists a row which is covered by only one column, both the column and the rows covered by the column are removed from the input matrix and the column is added to the final solution. For example, let Figure 2.1 on page E-5 be the input matrix for an SCP. Notice that row 2 is only covered by column 3. Hence, any solution to this problem must contain column 3. This column can be removed from the matrix along with any other rows that it covers. Figure E.8 shows the new input matrix to the SCP. When the search is complete, column 3 and its cost are added to the solution.

| | | Sets | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | 0 | 1 | 2 | 4 | 5 | 6 | 7 |
| Vertices | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 1 |
| | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| | 3 | 0 | 1 | 0 | 1 | 0 | 1 | 1 |
| | 4 | 0 | 0 | 0 | 1 | 1 | 1 | 0 |
| | 5 | 1 | 1 | 0 | 0 | 0 | 1 | 0 |
| | | 4 | 7 | 5 | 3 | 2 | 6 | 5 |
| | | Costs | | | | | | |

Figure E.8. Application of a Reduction Technique

3. Reduction #3: Any row which is dominated by another row in the matrix may be removed. "Let $V_i = \{j \mid r_i \in S_j\}$. Then, if $\exists p, q \in \{1, \ldots, M\}$ with $V_p \subseteq V_q$, $r_q$ may be deleted from $R$, since any set that covers $r_p$ must also cover $r_q$, i.e., $r_q$ is dominated by $r_p$" (17:40). Consider the two rows of the matrix in Figure E.9. Any

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| row $Q$: | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 |
| row $P$: | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 |

Figure E.9. Example Rows for SCP Reduction #3

column chosen to cover row $P$ must also cover row $Q$ since row $P$ is a subset of row $Q$; therefore, row $Q$ may be removed.

4. Reduction #4: Any column (set) which is dominated by another column (set) in the matrix may be removed. "If, for some family of sets $\overline{\mathcal{L}} \subset \mathcal{L}$ we have $\bigcup_{S_j \in \overline{\mathcal{L}}} S_j \supseteq S_k$ and $\sum_{S_j \in \overline{\mathcal{L}}} c_j \leq c_k$ for any $S_k \in \mathcal{L} - \overline{\mathcal{L}}$, then $S_k$ may be deleted from $\mathcal{L}$ since it is dominated by $\bigcup_{S_j \in \overline{\mathcal{L}}} S_j$" (17:40). Consider the two columns in Figure E.10. Column $Q$ is a subset

| | $P$ | $Q$ |
|---|---|---|
| | 1 | 1 |
| | 0 | 0 |
| | 1 | 1 |
| | 1 | 1 |
| | 0 | 0 |
| | 1 | 1 |
| | 0 | 0 |
| | 1 | 0 |
| | 1 | 1 |
| Costs | 5 | 6 |

Figure E.10. Example Columns for SCP Reduction #4

of $P$ and has a greater cost. Since, column $P$ covers the same rows as column $Q$ and has a lesser cost, it dominates column $Q$ and column $Q$ can be removed.

*E.6.5 Search Table Construction* The branch-and-bound search described in the previous section requires a significant amount of bookkeeping. In fact, it could be argued that all optimal search techniques are elaborate bookkeeping exercises (41). The branch-and-bound algorithm must store the traversed states so they can be recalled during the backtracking phase. Furthermore, it is desirable to choose only those sets which actually

contribute to the solution. For instance, in Figure 2.1, suppose the search algorithm has chosen sets {0, 1} to cover rows {0, 1, 3, 5}. It is pointless to choose set {2} since it will not cover any rows not already covered by sets {0, 1}. Therefore, the efficiency of the search process is improved if there exists some method to choose the next set that covers rows not already covered.

Christofides (17:41) suggests the construction of a table to assist in the bookkeeping and selection of the next set. The table for the matrix of Figure 2.1 is shown in Figure E.11.

| | | Blocks | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | 0 | | | | | 1 | | | | 2 | | | | 3 | | | 4 | | | 5 |
| | | Columns | | | | | | | | | | | | | | | | | | | |
| | | 5 | 0 | 7 | 2 | 1 | 5 | 0 | 2 | 7 | 4 | 7 | 6 | 1 | 5 | 4 | 6 | 0 | 6 | 1 | 3 |
| | 0 | 1 | 1 | 1 | 1 | 1 | | | | | | | | | | | | | | | |
| | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | | | | | | | | | | | |
| Rows | 3 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | | | | | | | |
| | 4 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | | | | |
| | 5 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | |
| | 2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| | | 2 | 4 | 5 | 5 | 7 | 2 | 4 | 5 | 5 | 3 | 5 | 6 | 7 | 2 | 3 | 6 | 4 | 6 | 7 | 8 |

Costs

Figure E.11. Table for Figure 2.1

The algorithm to build the table defines a block for each vertex (row) of the matrix. All sets (columns) covering a particular vertex are contained in the block for that vertex. A search algorithm which selects one set from each block is guaranteed to cover all the vertices. If, in addition to just selecting sets from the blocks, the search algorithm keeps track of the vertices already covered, the algorithm could skip blocks which correspond to vertices already covered. The search progresses from left to right in the table continually selecting and marking one set from each block as necessary. If the algorithm must backtrack, it regresses from right to left until it has found a block that can be further expanded. Notice, also, that the sets within each block are ordered in ascending order. This ordering, in most cases, decreases the number of expanded nodes in the search tree. The worst case,

of coarse, requires that all sets be checked before the optimal solution is found. As stated, the purpose of the table is to assist the search in the bookkeeping and selection of the next set.

Given the table of Figure E.11, a search is conducted. Figure E.12 illustrates the number of nodes expanded in the search tree when the only criteria for backtracking is the existence of a cover. It is included to illustrate the dramatic affect of applying the previously described reductions and dominance test. The matrix resulting from the application of the reductions is shown in Figure E.13 and the search of this reduced matrix is shown in Figure E.14. Clearly, the reductions can dramatically decrease the amount of time required to search the matrix. Note that this example is ideal and that not all input matrices are reducible.

The next two sections describe the dominance and lower-bound tests. Both tests are designed to limit the number of nodes expanded in the search tree. An example search tree of the nonreduced matrix is given for the dominance test.

*E.6.6 Dominance Testing* Christofides describes two elimination procedures which potentially improve the efficiency of the search. The first procedure is a dominance test. Let the state of the SCP search be defined by the tuple $\{E, B, z, \hat{B}, \hat{z}\}$ where:

- $E$ represents the set of rows currently covered in the search,
- $B$ represent the set of columns which cover the rows in $E$,
- $\hat{B}$ represent the set of covering columns for the best solution obtained thus far, and
- $z$ and $\hat{z}$ represent the cost of the covering columns contained in $B$ and $\hat{B}$, respectively.

As the search progresses, duplicate states are likely to occur in the search tree. A dominance test for the SCP saves limited information about the current state and uses it to compare to future states. Assume the algorithm selects column $S_j^k$ from the table for inclusion in the cover and that previous states were saved. If $E \cup S_j^k \subseteq E_{\text{previous}}$ and $z + c_j^k \geq z_{\text{previous}}$, the algorithm can immediately backtrack since the addition of $S_j^k$ can not result in a better cover than already obtained (17:44–45). Figure E.15 is a search of the original, unreduced matrix using a dominance test.

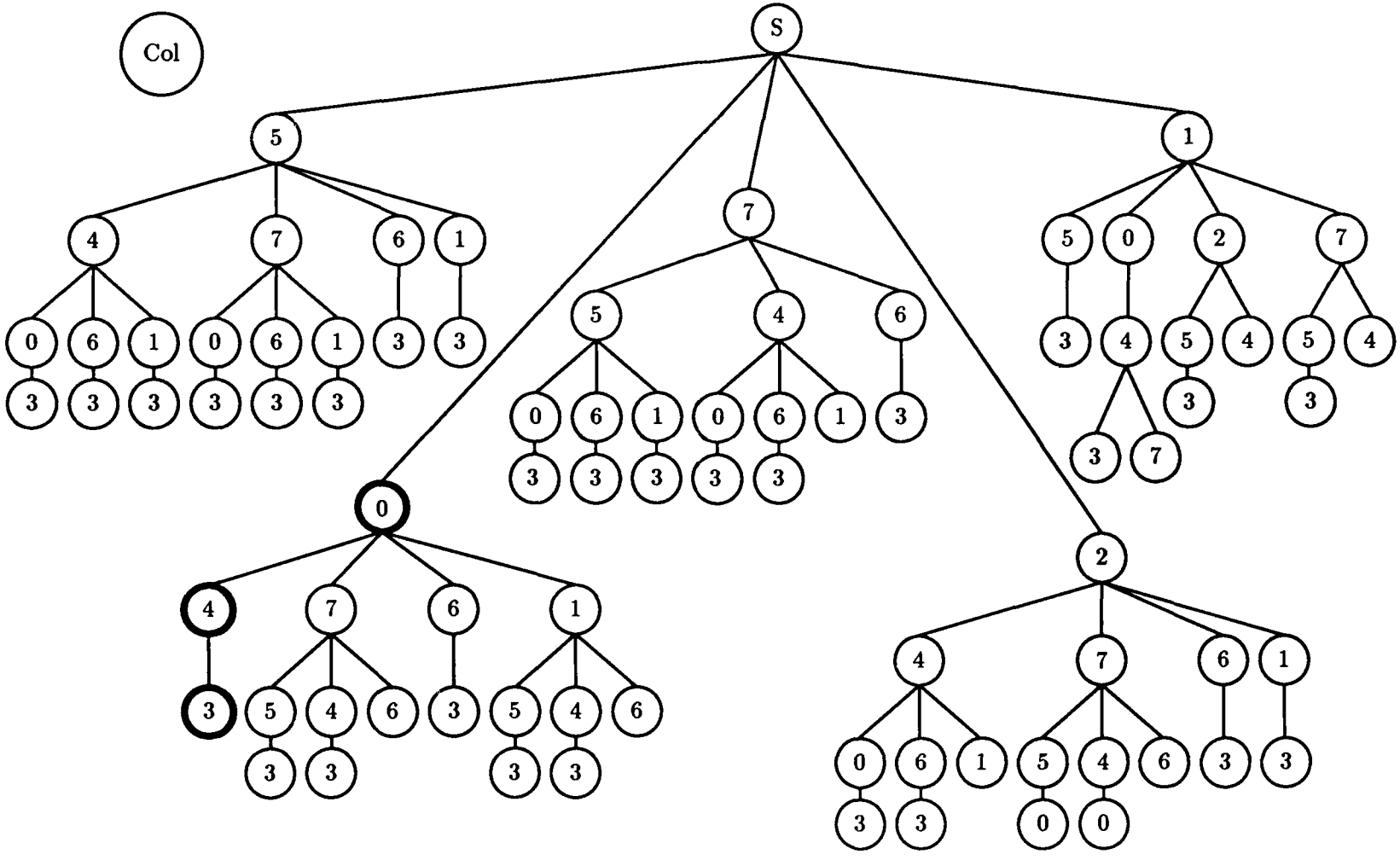



Figure E.12. Full Search of the Table in Figure E.11

| | | Sets | | | | | |
|---|---|---|---|---|---|---|---|
| | | 0 | 1 | 4 | 5 | 6 | 7 |
| | 1 | 1 | 0 | 0 | 1 | 0 | 1 |
| Vertices | 3 | 0 | 1 | 1 | 0 | 1 | 1 |
| | 4 | 0 | 0 | 1 | 1 | 1 | 0 |
| | 5 | 1 | 1 | 0 | 0 | 1 | 0 |
| | | 4 | 7 | 3 | 2 | 6 | 5 |
| | | Costs | | | | | |

Figure E.13. Reduced 0–1 Matrix of Figure 2.1

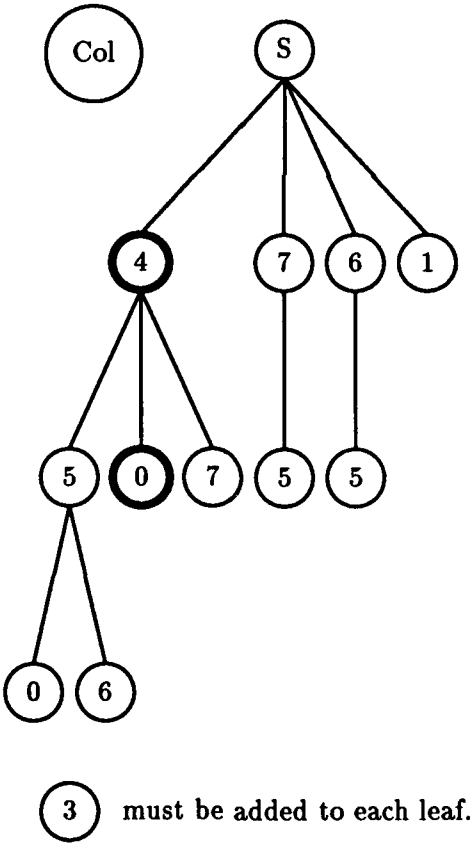



Figure E.14. Full Search Tree of Reduced Matrix

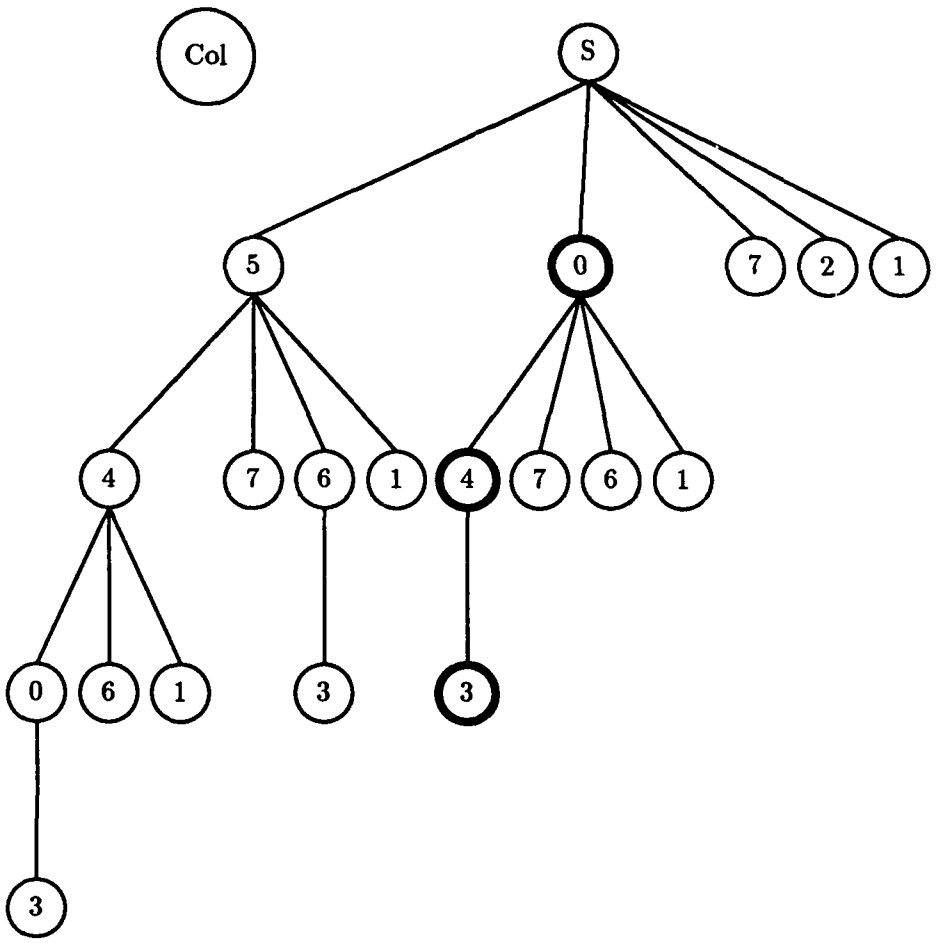



Figure E.15. SCP Search with Dominance Testing

*E.6.7 Lower Bound Computation* Christofides' second elimination procedure is a dynamic programming algorithm for the computation of a lower bound. "At some stage of the search, given by $B', E', z'$, and where block $k$ is the next block to be considered, a lower bound $h$ on the minimum value of $z$, can be calculated and used to limit the tree search" (17:44–45). The first step in the lower bound computation is to construct two matrices, $D$ and $D'$, as shown in Figure E.16. Let $M$ represent the number of rows to be covered in the 0–1 SCP matrix and $|E'|$ represent the number of rows already covered. Consider the next row to be covered, $r_j$, it can not be covered unless some column, $S_j^k$, is chosen from block $j$ of the table. For each $r_j$, "construct a row for matrix $D = [d_{qs}]$ and a row for a second matrix $D' = [d'_{qs}]$, where $d_{qs}$ is the number of elements in set $S_s^q$ and $d'_{qs}$ is its cost. In addition, append an extra row, $\theta$ say, to $D$ and $D'$ with $d_{\theta s} = s$ for all $s = 0, 1, \ldots, M - |E'|$ and $d'_{\theta s} = s \times \min[c_j^i/|S_j^i|]$, the minimum being taken over all sets $S_j^i$ with $r_i \notin E'$" (17:45). Suppose the search of the 0–1 SCP matrix of Figure 2.1 has just started ($M = 6$ and $|E'| = 0$). Computation of the lower bound at this state is meaningless since the lower bound will always be lower than the starting cost of infinity. Since this is just an illustration, the $D$ and $D'$ matrices for $M = 6$ and $|E| = 0$ are computed anyway and shown in Figure E.16.

$D$

| Rows | | | | | | | |
|---|---|---|---|---|---|---|---|
| | 0 | 3 | 3 | 3 | 3 | 3 | 3 |
| | 2 | 1 | 0 | 0 | 0 | 0 | 0 |
| | $\theta$ | 0 | 1 | 2 | 3 | 4 | 5 |

$D'$

| Rows | | | | | | | |
|---|---|---|---|---|---|---|---|
| | 0 | 2 | 4 | 5 | 5 | 7 | $\infty$ |
| | 2 | 8 | $\infty$ | $\infty$ | $\infty$ | $\infty$ | $\infty$ |
| | $\theta$ | 0 | 1 | 2 | 3 | 4 | 5 |

Figure E.16. $D$ and $D'$ Matrices for Figure 2.1

Notice that the matrices have only three rows and are rectangular. The sets in the first block of the table shown in Figure E.11 must be included since the first row is uncovered. The sets in this first block just happen to cover all rows in the adjacency matrix except for the last row. Hence, only the first and the last rows (Rows 0 and 2) are added to $D$ and $D'$. Some values of the rows may be undefined if that row's block does not have $M - |E'|$ columns. These undefined values are set to zero and infinity in $D$ and $D'$ respectively so that the resultant matrices are rectangular (17:45). The 'min' value used

to generate the final row of the $D'$ matrix is computed as follows:

$$\min[c_j^i/|S_j^i|] = \min[\frac{2}{3}, \frac{4}{3}, \frac{5}{3}, \frac{5}{3}, \frac{7}{3}, \frac{2}{2}, \frac{4}{2}, \frac{5}{1}, \frac{5}{2}, \frac{3}{2}, \frac{5}{1}, \frac{6}{3}, \frac{7}{2}, \frac{2}{1}, \frac{3}{1}, \frac{6}{2}, \frac{4}{1}, \frac{6}{1}, \frac{7}{1}, \frac{8}{1}] = 1$$

To compute the lower bound, the algorithm chooses one entry from each row of $D$ and $D'$ so that the sum of the entries covers the uncovered rows and the corresponding cost is minimum (17:45):

$$\sum_{q=1}^{\theta} d_{qs} \geq M - |E'| \tag{E.3}$$

$$\sum_{q=1}^{\theta} d'_{qs} \tag{E.4}$$

> Let $g_\rho(v)$ be the maximum number of elements that can be covered using the first $\rho$ rows of $D$ (i.e., only $\rho$ of the blocks in the subproblem), and whose total cost does not exceed $v$. $g_\rho(v)$ can then be calculated iteratively as:
>
> $$g_\rho(v) = \max_{s=1,\ldots,f} [d_{\rho s} + g_{\rho-1}(v - d'_{\rho s})], \tag{E.5}$$
>
> where $g_0(v)$ is initialized to 0 for all $v$. The lowest value, $v^*$, of $v$, for which $g_\theta(v) \geq M - |E'|$ is then the required lower bound, $h$. (17:45–46)

This concludes the detailed explanation of the SCP. Much of this information is contained in Christofides (17:Chapter 3). Other methods are certainly available and many are explained in the literature (18, 7, 52, 27, 43).

### *E.7 Summary*

This appendix presents examples of NP-complete problems and a detailed explanation of the SCP. These examples are typical of NP-complete problems and serial solutions to these problems are well known and documented for specific cases as well as for the general case (18, 4, 26, 17, 13, 29, 5). Furthermore, some work has been performed involving parallel solutions (56, 48, 23, 46, 50, 25). Of the previous examples, emphasis is placed on the SCP because of its general applicability to many different problems.

*Bibliography*


1. Abdelrahman, Tarek S. and Trevor N. Mudge. "Parallel branch and bound algorithms on hypercube multiprocessors." In *The Third Conference on Hypercube Concurrent Computers and Applications, Volume 2*, pages 1492–1499, The Association for Computing Machinery, 1988.
2. Adam, John A. and Mark A. Fischetti. "SDI: The Grand Experiment," *IEEE Spectrum*, *22*(9) (September 1985).
3. Adam, John A. and Paul Wallich. "Mind-Boggling Complexity," *IEEE Spectrum*, *22*(9) (September 1985).
4. Aho, Alfred B., John E. Hopcroft and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Reading, Massachusetts: Addison-Wesley, 1974.
5. Appel, Kenneth and Wolfgang Haken. "The Solution of the Four-Color-Map Problem," *Scientific American*, *237*(4):108–121,152 (October 1977).
6. Bailor, Paul D., Gary B. Lamont, and Thomas C. Hartrum. *A Conceptual Framework for Parallel Software Development*. Technical Report, Wright-Patterson Air Force Base, OH: Air Force Institute of Technology, January 1988. Written for the Department of Electrical and Computer Engineering.
7. Balas, Egon and Manfred W. Padberg. "On the Set-Covering Problem," *Operations Research*, *20*:1152–1162 (1972).
8. Beard, R. Andrew and Gary B. Lamont. "AFIT/ENG Intel Hypercube Quick Reference Manual." Version 1.1. Written for the Department of Electrical and Computer Engineering, February 1990.
9. Bellman, Richard E. and Stuart E. Dreyfus. *Applied Dynamic Programming*. Princeton, NJ: Princeton University Press, 1962.
10. Booch, Grady. *Software components with Ada*. Menlo Park, CA: The Benjamin/Cummings Publishing Company, Inc., 1987.
11. Borland International. *Turbo C Reference Guide, Version 2.0*, 1988.
12. Borland International. *Turbo C User's Guide, Version 2.0*, 1988.
13. Brassard, Giles and Paul Bratley. *Algorithmics: Theory and Practice* (First Edition). Englewood Cliffs, New Jersey: Prentice Hall, 1988.
14. Carpenter, Barry Austin. *Implementation and Performance Analysis of Parallel Assignment Algorithms on a Hypercube Computer*. MS thesis, Air Force Institute of Technology, Wright-Patterson Air Force Base, Ohio, December 1987.
15. Carriero, Nicholas and David Gelernter. "How to write parallel programs: A guide to the perplexed," *ACM Computing Surveys*, *21*(3):323–357 (September 1989).
16. Chandy, K. Mani and Jayadev Misra. *Parallel Program Design: A Foundation*. Reading, Massachusetts: Addison-Wesley, 1988.

17. Christofides, Nicos. *Graph Theory: An Algorithmic Approach.* London, England: Academic Press, 1975.

18. Christofides, Nicos and S. Korman. "A Computational Survey of Methods for the Set Covering Problem," *Management Science*, *21*(5):591–599 (January 1975).

19. Coffman, Edward G., Jr. and Peter J. Denning. *Operating System Theory.* Englewood Cliffs, New Jersey: Prentice-Hall, 1973.

20. Computer Innovations, Inc. *Optimizing C86 User's Manual*, May 1985. Document Version 2.3, Software Version 2.3.

21. Cvetanovic, Zarka. "The Effects of Problem Partitioning, Allocation, and Granularity on the Performance of Multiple-Processor Systems." In *IEEE Transactions on Computers*, pages 421–432, IEEE, April 1987.

22. Ercal, F., J. Ramanujam and P. Sadayappan. "Task Allocation onto a Hypercube by Recursive Mincut Bipartitioning." In *The Third Conference on Hypercube Concurrent Computers and Applications: Volume 1*, pages 210–221, The Association for Computing Machinery, 1988.

23. Felten, Edward W. "Best-First Branch-and-Bound on a Hypercube." In *The Third Conference on Hypercube Concurrent Computers and Applications, Volume 2*, pages 1500–1504, The Association for Computing Machinery, 1988.

24. Flynn, M. J. "Very high-speed computers," *Proceedings of the IEEE*, *54*(12):1901–1909 (December 1966).

25. Fox, G., M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker. *Solving Problems on Concurrent Processors.* Englewood Cliffs, New Jersey: Prentice-Hall, 1988.

26. Garey, Michael R. and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* San Francisco, California: W. H. Freeman and Company, 1979.

27. Garfinkel, R. S. and G. L. Nemhauser. "The Set-Partitioning Problem: Set Covering With Equality Constraints," *Operations Research*, *17*:848–856 (1969).

28. Hoare, C. A. R. *Communicating Sequential Processes.* Englewood Cliffs, New Jersey: Prentice-Hall, 1985.

29. Horowitz, Ellis and Sartaj Sahni. *Fundamentals of Computer Algorithms.* Rockville, Maryland: Computer Science Press, 1978.

30. Horowitz, Ellis and Sartaj Sahni. *Fundamentals of Data Structures.* Rockville, Maryland: Computer Science Press, 1983.

31. Hwang, Kai and Faye A. Briggs. *Computer Architecture and Parallel Processing.* New York: McGraw-Hill, 1984.

32. Intel. Intel iPSC/2 promotional literature. EENG 689 class handout. Department of Electrical and Computer Engineering, Air Force Institute of Technology (AU), Wright-Patterson Air Force Base, OH, 45433. Spring Quarter, 1989.

33. Intel Corporation. *iPSC/2 System Overview*, November 1986. Order Number: 310610–001.

34. Intel Corporation. *iPSC/2 C Language Reference Manual*, November 1988. Order Number: 311567–002.

35. Intel Corporation. *iPSC/2 C Programmer's Reference Manual*, August 1988. Order Number: 311017–003.

36. Intel Corporation. *iPSC/2 System Administrator's Guide*, August 1988. Order Number: 311014–003.

37. Intel Corporation. *iPSC/2 User's Guide*, March 1989. Order Number: 311532–003.

38. Jamieson, Leah H., Dennis Gannon and Robert J. Douglass. *The Characteristics of Parallel Algorithms*. Cambridge, Massachusetts: The MIT Press, 1987.

39. Kernighan, Brian W. and Dennis M. Ritchie. *The C Programming Language*. Englewood Cliffs, New Jersey: Prentice-Hall, 1978.

40. Lai, T-H and S. Sahni. "Anomalies in parallel branch-and-bound algorithms," *CACM*, *27*(6):594–602 (March 1984).

41. Lamont, Gary B. EENG 685 class lecture. Department of Electrical and Computer Engineering, Air Force Institute of Technology (AU), Wright-Patterson Air Force Base, OH, 45433. Fall Quarter, 1988.

42. Lamont, Gary B. and Donald J. Shakley. "Parallel Expert System Search Techniques for a Real-Time Application." In *The Third Conference on Hypercube Concurrent Computers and Applications, Volume 2*, pages 1352–1359, The Association for Computing Machinery, 1988.

43. Lemke, C. E., H. M. Salkin and K. Spielberg. "Set Covering By Single-Branch Enumeration With Linear-Programming Subproblems," *Operations Research*, *19*(4):998–1022 (July-August 1971).

44. Ma, Richard P., Fu-Sheng Tsung, and Mae-Hwa Ma. "A Dynamic Load Balancer for a Parallel Branch and Bound Algorithm." In *The Third Conference on Hypercube Concurrent Computers and Applications, Volume 2*, pages 1505–1513, The Association for Computing Machinery, 1988.

45. Nemhauser, George L. *Introduction to Dynamic Programming*. Princeton, NJ: John Wiley and Sons, Inc., 1966.

46. Pangas, Roy P. and Wooster, E. Daniels. "Branch-and-bound algorithms on a hypercube." In *The Third Conference on Hypercube Concurrent Computers and Applications, Volume 2*, pages 1514–1519, The Association for Computing Machinery, 1988.

47. Pearl, Judea. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Reading, Massachusetts: Addison-Wesley, 1984.

48. Pettey, Chrisila and Michael R. Leuze. "Parallel Placement of Parallel Processes." In *The Third Conference on Hypercube Concurrent Computers and Applications: Volume 1*, pages 232–238, The Association for Computing Machinery, 1988.

49. Pressman, Roger S. *Software Engineering: A Practitioners Approach*. New York: McGraw-Hill, 1987.

50. Quinn, Michael J. *Designing Efficient Algorithms for Parallel Computers*. New York: McGraw-Hill, 1987.

51. Quinn, Michael J. "Analysis and Implementation of Branch-and-Bound Algorithms on a Hypercube Multicomputer," *IEEE Transactions on Computers*, *39*(3):384–387 (March 1990).

52. Roth, R. "Computer Solutions to Minimum-Cover Problems," *Operations Research*, *17*:455–465 (1969).

53. Schwan, Karsten, John Gawkowski and Ben Blake. "Process and workload migration for a parallel branch-and-bound algorithm on a hypercube multicomputer." In *The Third Conference on Hypercube Concurrent Computers and Applications, Volume 2*, pages 1520–1530, The Association for Computing Machinery, 1988.

54. Stanat, Donald F. and David F. McAllister. *Discrete Mathematics in Computer Science*. Englewood Cliffs, New Jersey: Prentice-Hall, 1977.

55. Stone, Harold S. *High-Performance Computer Architecture*. Reading, Massachusetts: Addison-Wesley, 1987.

56. Wah, Benjamin W., Guo-jie Li and Chee Fen Yu. "Multiprocessing of Combinatorial Search Problems," *Computer*, *18*(6):93–108 (June 1985).

## *Vita*

Captain Ralph A. Beard [redacted]. He received his high school equivalency in Great Falls, Montana in 1978 while enlisted in the Air Force. From 1978 to 1982 he served as a missile maintenance technician and maintenance controller in the 341st Strategic Missile Wing. He was accepted into the Airman's Education and Commissioning Program (AECP) in 1982 and attended the University of Florida where he graduated with a Bachelor of Science in Electrical Engineering in September 1985. Upon graduation, he attended USAF Officer's Training School from which he received his commission. He then served as a design engineer in the Avionics Laboratory of the Air Force Wright Aeronautical Labs until 1988. He entered the School of Engineering, Air Force Institute of Technology in June 1988 to pursue a Master of Science in Computer Engineering.

[redacted]

# REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

| | |
|---|---|
| 1a. REPORT SECURITY CLASSIFICATION<br>Unclassified | 1b. RESTRICTIVE MARKINGS |
| 2a. SECURITY CLASSIFICATION AUTHORITY | 3. DISTRIBUTION/AVAILABILITY OF REPORT<br>Approved for public release; distribution unlimited |
| 2b. DECLASSIFICATION/DOWNGRADING SCHEDULE | |
| 4. PERFORMING ORGANIZATION REPORT NUMBER(S)<br>AFIT/GCE/ENG/90M-1 | 5. MONITORING ORGANIZATION REPORT NUMBER(S) |
| 6a. NAME OF PERFORMING ORGANIZATION<br>School of Engineering<br>6b. OFFICE SYMBOL (If applicable)<br>AFIT/ENA | 7a. NAME OF MONITORING ORGANIZATION |
| 6c. ADDRESS (City, State, and ZIP Code)<br>Air Force Institute of Technology (AU<br>Wright-Patterson AFB, OH 45433-6583 | 7b. ADDRESS (City, State, and ZIP Code) |
| 8a. NAME OF FUNDING/SPONSORING ORGANIZATION<br>8b. OFFICE SYMBOL (If applicable)<br>SDIO S/PI | 9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER |
| 8c. ADDRESS (City, State, and ZIP Code)<br>Room 1E149<br>The Pentagon<br>Washington DC 20301-7100 | 10. SOURCE OF FUNDING NUMBERS<br>PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO / WORK UNIT ACCESSION NO. |

11. TITLE (Include Security Classification)
Determination of Algorithm Parallelism in NP-Complete Problems for Distributed Architectures

12. PERSONAL AUTHOR(S)
Beard, Ralph Andrew/Capt/USAF

| 13a. TYPE OF REPORT | 13b. TIME COVERED | 14. DATE OF REPORT (Year, Month, Day) | 15. PAGE COUNT |
|---|---|---|---|
| MS Thesis | FROM _____ TO _____ | 1990 March 5 | 232 |

16. SUPPLEMENTARY NOTATION
Thesis Advisor: Gary B. Lamont, Professor; Department of Electrical and Computer Engineering; Air Force Institute of Technology

| 17. COSATI CODES | | |
|---|---|---|
| FIELD | GROUP | SUB-GROUP |
| | | |
| | | |

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
NP-complete, Set Covering Problem, SCP, Parallel Processing, Parallel Algorithms

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

The purpose of this research is to explore the methods used to parallelize NP-complete problems and the degree of improvement that can be realized using a distributed parallel processor to solve these combinatoric problems.

Common NP-complete problem characteristics such as a priori reductions, use of partial state information, and inhomogeneous searches are identified and studied. The set covering problem (SCP) is implemented for this research because many applications such as information retrieval, task scheduling, and VLSI expression simplification can be structured as an SCP problem. In addition, its generic NP-complete common characteristics are well documented and a parallel implementation has not been reported.

Parallel programming design techniques involve decomposing the problem and developing the parallel algorithms. The major components of a parallel solution are developed in a four phase process. First, a meta-level design

| | |
|---|---|
| 20. DISTRIBUTION/AVAILABILITY OF ABSTRACT<br>☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS | 21. ABSTRACT SECURITY CLASSIFICATION<br>Unclassified |
| 22a NAME OF RESPONSIBLE INDIVIDUAL<br>Ralph A. Beard, Capt, USAF | 22b. TELEPHONE (Include Area Code)<br>(513) 255-3030<br>22c. OFFICE SYMBOL<br>AFIT/ENA |

is accomplished using an appropriate design language such as UNITY. Then, the UNITY design is transformed into an algorithm and implementation specific to a distributed architecture. Finally, a complexity analysis of the algorithm is performed.

The a priori reductions are divide-and-conquer algorithms; whereas, the search for the optimal set cover is accomplished with a branch-and-bound algorithm. The search utilizes a global best cost maintained at a central location for distribution to all processors. Three methods of load balancing are implemented and studied: coarse grain with static allocation of the search space, fine grain with dynamic allocation, and dynamic load balancing.

A serial and three SCP parallel algorithms were implemented and executed on an iPSC/2 computer. Tests on large SCP problems indicate limited speedup over the serial program with the coarse grain version using static allocation and improved speedup with the fine grain version using dynamic allocation. The use of dynamic load balancing further improves the speedup and led to a super-linear speedup.